City University of New York (CUNY)
CUNY Academic Works

Open Educational Resources | Queensborough Community College

2021

Introduction to Computers and Programming using Python: A Project-based Approach

Esma Yildirim
CUNY Queensborough Community College

Daniel Garbin
CUNY Queensborough Community College

Mathieu Sassolas
CUNY Queensborough Community College

Kwang Hyun Kim
CUNY Queensborough Community College

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/qb_oers/170
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Introduction to Computers and Programming using Python: A Project-based Approach

Daniel Garbin   Kwang Hyun Kim   Mathieu Sassolas   Esma Yildirim

2021-01-26

Department of Mathematics and Computer Science
Queensborough Community College
City University of New York

Student version

This Open Education Resource is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License.

Contents

Foreword iii

I Concept review 1

L1 Variables 3

L2 Input and Ouput 9

L3 Numbers and arithmetic 15

L4 Strings 23

L5 Conditions 35

L6 Loops 45

L7 Lists, Tuples, Dictionaries 53

L8 Functions 67

II Projects 83

P1 The Motion of the Vertical Projectile 85

P2 Linear and Quadratic Equations 89

P3 Image Processing 97

P4 Simple Operations with Fractions 109

P5 Time Measurement and Dates 119

P6 Grade Management with Pandas 129

P7 Descriptive Statistics and Histogram of Frequencies 139

P8 Emotion Analysis 147

P9 Dynamics on Functions 159

P10 The Game of Tic-Tac-Toe **167**

P11 A Function-Based Role Playing Game **175**

P12 Voting Systems **195**

P13 Protein Translation **211**

Foreword

0.1 How to use this OER?

This Open Educational Resource has been designed to give freedom to the instructor, both in format and topics ultimately used throughout the course. While we provide 13 turnkey projects, it is only expected that 3 or 4 are used over the course of a semester. And all projects are provided both as textual instructions (the student version of this OER) and Jupyter Notebooks (one with and one without the solutions). It is up to the instructor to choose the most efficient platform according to the context of the class and technical constraints.

0.1.1 The projects

Not all projects are created equal: they all assume some degree of proficiency in certain aspects of programming, and provide practice in other aspects. A list of all concepts used in projects is provided in Table 1, and a brief description of each project is given below.

P1: The Motion of the Vertical Projectile. *Physics*
This project uses the arithmetical and mathematical features of Python to implement the formulas for the vertically thrown projectile.

P2: Linear and Quadratic Equations. *Mathematics*
In this project we use Python to solve linear and quadratic equations. Emphasis is put on the special cases that need to be detected and dealt with using conditionals.

P3: Image Processing. *Graphics*
In this project we use the PIL library to manipulate images pixel by pixel. We implement changes of color and transparency traversing the 2-dimensional array of the image. A more advanced topic blurs the image by using a vicinity of the pixels. These changes can be performed on any image chosen by the students.

P4: Simple Operations with Fractions. *Mathematics*
The goal of this project is to code basic algebra on rational numbers. It starts with fraction simplification (using Euclid's Algorithm for the GCD) and then gets into the 4 basic operations.

P5: Time Measurement and Dates. *Computer applications*
This project introduces the Time library. We use its basic features to measure elapsed time, wait some time, then implement a conversion from time as seconds from *epoch* into a real date.

P6: Grade Management with Pandas. *Statistics*
In this project, we use the Pandas library to read and write Excel spreadsheet. We use it to build a grade management system.

#	Title	Prerequisites	Concepts covered
P1	*The Motion of the Vertical Projectile*	Input/output, lists, conditionals, loops, functions	Working with math formulas
P2	*Linear and Quadratic Equations*	Input/output, conditionals	Working with math formulas
P3	*Image Processing*	Functions, conditionals	Loops, list and tuple access
P4	*Simple Operations with Fractions*	Input/output, lists, conditionals, loops	Functions
P5	*Time Measurement and Dates*	Input/output, functions, conditionals	Loops
P6	*Grade Management with Pandas*	Input/output, lists, conditionals, functions	Pandas, vector operations
P7	*Descriptive Statistics and Histogram of Frequencies*	Loops, lists	Statistics library functions, bar charts
P8	*Emotion Analysis*	Loops, conditionals, input/output	Lists, dictionaries, bar charts, natural language processing
P9	*Dynamics on Functions*	Input/output, lists, conditionals, loops, functions	Recursion and iteration
P10	*The Game of Tic-Tac-Toe*	Input/output, lists, conditionals, loops, functions	Basic game design
P11	*A Function-Based Role Playing Game*	Variables, input/output	Functions, conditionals, dictionaries, the importance of comments
P12	*Voting Systems*	Lists, basic loops, conditionals	Dictionaries and looping on dictionaries
P13	*Protein Translation*	Lists, loops, functions, input/output, conditionals	Dictionaries

Table 1: Programming concepts in the projects.

P7: Descriptive Statistics and Histogram of Frequencies. *Statistics*
In this project, we calculate measures of central tendency and dispersion. We also use the Matplotlib and Seaborn libraries to create histograms using bar graphs.

P8: Emotion Analysis. *Literature*
This project uses word dictionaries and dedicated Python libraries to analyze and compare the emotions conveyed by books. Two books are analyzed in the project but students are encouraged to choose their own favorite book and extract the emotion information from it as well.

P9: Dynamics on Functions. *Computer Science*
This project focuses on composition of functions and applies it to the computational verification of the Collatz conjecture.

P10: The Game of Tic-Tac-Toe. *Games*
In this project, we implement the 4-by-4 version of the Tic-Tac-Toe game.

P11: A Function-Based Role Playing Game. *Games*
In this project, we implement a simple donjon in an RPG. This implementation relies on function calling (which can be recursive) and does not require to use any loop.

P12: Voting Systems. *Political science*
This project surveys different mode of elections: first-past-the-post, Borda, Condorcet, two rounds, instant runoff. For each of these, a list of preferences is provided and the goal is to find the winner according to the voting systems.

P13: Protein Translation. *Biology*
In this project, we implement a simulation of the translation from DNA to proteins that occur in living cells. From a file containing the DNA and a file of RNA amino-acid codons, we generate a dictionary of proteins.

0.1.2 A typical course sequence

Before getting onto projects, we recommend introducing some basic concepts formally. The Lectures provide not only an introduction to the various programming constructs, but are to be used as a reference throughout the project. Whenever a concept is used for the first time in a project, a pointer to refer to the relevant Lecture is provided so that student get a refresher on the topic. Like projects, Lectures are provided both in this document and in Jupyter Notebook format.

To guide the choice of projects, here is a couple of suggestions. These sequences provide a good coverage of topics with an increasing project difficulty.

- 1. The Motion of the Vertical Projectile
  2. Image Processing
  3. Descriptive Statistics and Histogram of Frequencies
  4. Emotion Analysis

- 1. Linear and Quadratic Equations
  2. Grade Management with Pandas
  3. Protein Translation

- 4. A Function-Based Role Playing Game

- 1. The Motion of the Vertical Projectile
  2. Time Measurement and Dates
  3. Dynamics on Functions
  4. Voting Systems

- 1. Simple Operations with Fractions
  2. The Game of Tic-Tac-Toe
  3. Descriptive Statistics and Histogram of Frequencies
  4. Protein Translation *or* Voting Systems

0.2 Acknowledgements

This Open Education Resource was produced with funding from QCC CollegeNow program and CUNY Tutorcorps.

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

The authors would like to thank Mary-Ann Meyer and Maria Mercedes Franco for their ongoing support throughout the elaboration of this document.

Part I

Concept review

Lecture 1
Variables

Lecture contents

L1.1 Introduction 3
L1.2 Naming variables 3
L1.3 Defining and assigning variables 4
L1.3.1 Examples 4
L1.4 Using variables 5
L1.4.1 Examples 5
L1.5 Types 5
L1.5.1 An aside: Python's typing method 6
L1.6 Note on nomenclature: Variable, Name, or Object? 7

L1.1 Introduction

In a program, a **variable** is like a box to store data. It has a *name* and a *value*.

The name of the variable will be used in place of the value when coding. It will be replaced by its value at execution time. Using variables allows to create programs in an abstract way: the program concentrates on the *operations* that are performed on the data, not on the data itself, which is usually unavailable at the time of coding.

L1.2 Naming variables

The name is case sensitive; so variable `X` is a different variable than `x`, and `myVariable` is not the same as `Myvariable`.

The name of the variable can contain lowercase and uppercase characters, digits, as well as the underscore `_`, but it cannot start with a digit. The choice of the name of the variables is up to the programmer defining them, but one should keep in mind the following guidelines:

- Do not define several variables that only differ in case: it is a recipe for bugs when using one variable for the other.
- Choose a meaningful name that tells you at a glance what is the value that this variable contains. For example: `counter`, `firstName`, `nbWords`.
- Variable names cannot be keywords: so `if`, `for`, `while`, `return`, for example, should not be used as variable names. The complete list of Python keywords is below:

```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more␣
↪help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

The data can be of different kind: a number (with or without decimal point), some text... Data types will be discussed in Section L1.5.

L1.3 Defining and assigning variables

Assignment means giving a value to a variable. The first time a variable is assigned, it is also defined: beforehand it did not exist, so trying to use it will raise an error.

The syntax for an assignment is as follows: `<variable name> = <value>`. The value can be a calculated expression (this will be developed later).

L1.3.1 Examples

```
x = 42
X = 26 # Not the same as x
print("x and X:",x,X) # Prints the values of both x and X
myVariable = 10
print("[5] myVariable:",myVariable)  # Prints the current value (at line␣
↪5)
myVariable = 75 # Change of value for myVariable
print("[7] myVariable:",myVariable) # Prints the current value (at line 7)
myvariable = 42 # There is a typo: this defines a new variable instead of␣
↪changing the value of myVariable!
print("[9] myVariable:",myVariable) # Prints the current value (at line 9)
# print("y:",y) # Error: "NameError: name 'y' is not defined"
y = "Hello!"
print("y:",y) # No error
```

```
x and X: 42 26
[5] myVariable: 10
[7] myVariable: 75
[9] myVariable: 75
y: Hello!
```

Remark for Jupyter

In Jupyter, line 10 of the above block (`print("y:",y) "Hello!"`) will only produce an error if the block has not yet been executed. Otherwise variable `y` does exist, from the definition of line 11 executed in a previous execution of the block.
The kernel can be restarted in order to be able to execute a block "for the first time".

L1.4 Using variables

Whenever a variable name is used, it will be replaced by its *current* value. This is actually what is happening when we call `print` to see the value: it is first replaced by the values stored in the variable, and this value is printed out.

L1.4.1 Examples

```
x = 3
y = 5
z = 3+2*x # Note: PEMDAS applies
print("x, y, z:",x,y,z)
x = y # Now x is 5; z did not change
print("x, y, z:",x,y,z)
y = y+1 # Now y is 6; x did not change
print("x, y, z:",x,y,z)
```

```
x, y, z: 3 5 9
x, y, z: 5 5 9
x, y, z: 5 6 9
```

L1.5 Types

What happens with a variable depends on their **type**. For example, the number `3` is not the same as the string `"3"`. You cannot add number 1 to string `"3"`: it makes as much sense as trying to add number 1 to `"Hello!"`.

```
message = "3"
number = 3
addingNumber = number+1     # No problem
addingMessage = message+1  # Error: "TypeError: Can't convert 'int'␣
↪object to str implicitly"
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-3-503b622ba663> in <module>
  2 number = 3
  3 addingNumber = number+1 # No problem
```

```
->4 addingMessage = message+1  # Error: "TypeError: Can't convert 'int'␣
↪object to str implicitly"


TypeError: Can't convert 'int' object to str implicitly
```

Note however that + and * can still have a meaning with strings, but not the usual meaning of addition and multiplication that they have with numbers. See Lecture L4 for more details.

Python uses a pretty loose typing, so it is possible to have a variable that starts as a number but then is changed into a string. Just because it is possible does not mean it is desirable! (And other programming languages don't have this flexibility.) It is usually best to keep in mind what type of value the variable contains and use it as such all the time.

```
randomValue = 42
print(randomValue)
randomValue = randomValue+1 # No problem: it is a number
randomValue = "Goodbye!"    # Not a good idea
print(randomValue)
randomValue = randomValue+1 # Exact same line as line 3: TypeError
```

```
42
Goodbye!

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-4-7df054832887> in <module>
  4 randomValue = "Goodbye!"    # Not a good idea
  5 print(randomValue)
->6 randomValue = randomValue+1   # Exact same line as line 3: TypeError


TypeError: Can't convert 'int' object to str implicitly
```

L1.5.1 An aside: Python's typing method

Python use the *Duck typing* method, based on the following idea: "If it walks like a duck, and quacks like a duck, then it must be a duck".

So if a variable behaves like a number, it can be treated as a number for all useful purposes. In this context, Python does not actually try to know what is the type of a variable until it is time to use it. This is particularly important with integers and decimal-point numbers, which can in most cases be used interchangeably... unless they can't.

```
x = 5
x = 2.5 # Works with or without decimal point
y = 2*x+3
y = y*y-x
```

```
print(y)
```

```
61.5
```

```
x = 5
x = 2.5
for i in range(0,x):# Does not work: only integers are allowed:␣
↪"TypeError: 'float' object cannot be interpreted as an integer"
    print(i)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-1-c65c3110b7af> in <module>
  1 x = 5
  2 x = 2.5
->3 for i in range(0,x):# Does not work: only integers are allowed:␣
↪"TypeError: 'float' object cannot be interpreted as an integer"
  4     print(i)


TypeError: 'float' object cannot be interpreted as an integer
```

L1.6 Note on nomenclature: Variable, Name, or Object?

The term *variable* is used to denote two different things at the same time: the **name** of the variable and the **object** that is referred by that name, which is stored in the memory where the data resides.

In Python, the term *name* is commonly used, for example in the error that appears when a variable has not been defined: `NameError: name 'y' is not defined`.

Note that every value is an *object*: an integer or a float (these are called *primitive types*), a string, a list, or an instance of a class (in so-called *object-oriented programming*). In addition, objects need not have a name to exist. For example: `a = 5.3` uses the named object (or variable) `a` which is of `float` type and an unnamed or verbatim object `5.3` which is also of `float` type.

Lecture 2
Input and Ouput

Lecture contents

L2.1 Introduction 9
L2.2 The output operation 10
L2.2.1 Verbatim data 10
L2.2.2 Variable data 10
L2.2.3 Printing several values 11
L2.2.4 The `sep` and `end` options 11
L2.3 The input operation 12
L2.3.1 Conversion of inputs 12
L2.3.2 More string manipulation 13

L2.1 Introduction

The most basic tasks or operations of a programming language are the input and the output of data. The operation of input involves getting data or information. In the simplest case, the data will be entered by the user. This task is similar to the actions in the order below (which is really important):

1. ask someone a question
2. receive and remember the answer.

For example, the teacher asks the student “what day is today?” to which the student may reply ’Tuesday’. The teacher will then store this information (at least temporarily). Here are some further question/answer examples:

- Q: How long is this route? A: 230 km.
- Q: How much does the watermelon weigh? A: 3 kg.

The operation of output involves displaying or printing data. This task is similar to the actions in the order below:

1. display the context of the data (in other words, do not just print a value alone as it may be confusing);
2. display the data.

In the above, example the teacher may reply ‘Today is Tuesday’. Note that simply displaying the data ‘Tuesday’ without a proper context may be confusing.

For the sake of simplicity, we start by looking at the output before considering input.

L2.2 The output operation

The output operation involves the display of data in a given context. In Python this operation is done using the `print()` function.

Note: In this context, the function `print()` displays the content on the screen on the next available line. This is not to be confused with the content being sent for printing to a printer on paper.

L2.2.1 Verbatim data

The simplest examples consist of printing *verbatim data*. The data is placed inside a pair of single quotes e.g. `'Hello, there!'` (or double quotes `"Hello, there!"`).

```
print('Hello, world!')
```

```
Hello, world!
```

```
print('Today is Tuesday, our first class.')
```

```
Today is Tuesday, our first class.
```

Here is a more tedious example of printing a message using “character graphics”.

```
print('*    * ****** *      *       ****     *     *  ****  *    *  *')
print('*    * *      *      *      *    *     *   *  *    * *    *  *')
print('****** ****   *      *      *    *       *    *    * *    *  *')
print('*    * *      *      *      *    *       *    *    * *    *  *')
print('*    * *      *      *      *    *       *    *    * *    *   ')
print('*    * ****** ****** ******  ****        *     ****   ****   *')
```

```
*    * ****** *      *       ****     *     *  ****  *    *  *
*    * *      *      *      *    *     *   *  *    * *    *  *
****** ****   *      *      *    *       *    *    * *    *  *
*    * *      *      *      *    *       *    *    * *    *  *
*    * *      *      *      *    *       *    *    * *    *
*    * ****** ****** ******  ****        *     ****   ****   *
```

L2.2.2 Variable data

While some data is verbatim and does not change, other is flexible. To print variable data:

1. set a variable with a value (or datum);
2. print the variable.

```
s = 'Hello, world!'
print(s)
```

```
Hello, world!
```

The variable `s` is a string, i.e. an ordered collection of characters (simply put an array or sequence of characters).

L2.2.3 Printing several values

Print can display several values on the same line. These values are separated by a comma.
In the examples below, a comma separates the verbatim data from the variable.

```
a = 5
print('a =', a)
```

```
a = 5
```

```
pi = 3.1415
print('The value of PI is approximately ', pi, '.')
```

```
The value of PI is approximately  3.1415 .
```

```
weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',␣
↪'Saturday', 'Sunday']
print('The days of the week are: ', weekdays, '.')
```

```
The days of the week are:  ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday'] .
```

In the last example, the object weekdays is a `list` of `string` objects.

L2.2.4 The `sep` and `end` options

The `print()` function can take two optional arguments **`sep`** and **`end`**. They are given after the values that are to be printed and using the syntax `print(<values>,sep=<separator>,end=<endmark>)`

- `sep` determines what is to be printed between two comma separated values; by default it is a space.
- `end` determines what is to be printed after the last value; by default it is a new line.

```
print("a","b","c","d", sep="-")
```

```
a-b-c-d
```

```
print("Hello",end="")
print("Bye")
```

```
HelloBye
```

```
print(1,2,3,4,sep=",",end='!')
```

```
1,2,3,4!
```

L2.3 The input operation

The input operation is done by using the `input()` function. It requires:

- the name of the variable where the data will be stored;
- a context (usually as verbatim data), namely what we are interested to know from the user.

In the examples below, after each input the data is displayed using the `print` function.

```
s = input('Enter a greeting: ')
print('You entered: ', s)
```

```
Enter a greeting: Hello Jane!
You entered:  Hello Jane!
```

```
a = input('Enter an integer: ')
print('You entered: ', a)
```

```
Enter an integer: 34
You entered:  34
```

L2.3.1 Conversion of inputs

The input function sets the variable `a` with a `string` object. In this case, the user is asked for a number because the goal is to do computations with these `int` objects. This requires conversion to `int`. Here is what happens without conversion:

```
a = input('Enter an integer: ')
print('You entered: ', a)
b = input('Enter another integer: ')
print('You entered: ', b)
print('The sum of these integers is ', a+b)
```

```
Enter an integer: 3
You entered:  3
Enter another integer: 4
You entered:  4
The sum of these integers is  34
```

The code still works, but the sum `a+b` is rather strange. This is merely joining 2 strings a process also known as concatenation. If we tried `a-b` or `a/b` instead of `a+b`, it would not work on strings or text objects and produce an error.

The above code can be fixed by *converting* `a` and `b` into `int`:

```
a = input('Enter an integer: ')
a = int(a)
print('You entered: ', a)
b = input('Enter another integer: ')
```

```
print('You entered: ', b)
b = int(b)
print('The sum of these integers is ', a+b)
```

```
Enter an integer: 3
You entered:  3
Enter another integer: 4
You entered:  4
The sum of these integers is  7
```

There are 3 data type conversion functions:

- `int()`: to convert a `string` object to an `int`.
- `float()`: to convert a `string` object to a `float` point number (decimal representation).
- `str()`: to convert an `int` or `float` object to a `string`.

L2.3.2 More string manipulation

For more details about the operations that can be performed on `string`s, refer to Lecture L4.

Lecture 3
Numbers and arithmetic

Lecture contents

L3.1 Introduction 15
L3.2 Integers 15
L3.3 Floating point numbers 16
L3.4 Operations on numbers 16
L3.4.1 Addition, subtraction, multiplication 17
L3.4.2 Division 17
L3.4.3 Conversion between int and float 18
L3.4.4 Mixing integers and floats 19
L3.4.5 Other operations 19
L3.4.6 Precedence (a.k.a PEMDAS) 19
L3.4.7 Abbreviated operators 20

L3.1 Introduction

Numbers in Python, like in all computer languages, come in two flavors: **integers** and **floating point numbers** (also known as **floats**). These two ways of storing numbers are quite different from one another, and as such one must be careful not to confuse one for the other when manipulating number variables.

L3.2 Integers

Integers store whole numbers. In mathematical terms, numbers in $\mathbb{Z}$ can be stored as integers. Unlike other programming languages, there is no limit in the size of integers, except the size of the computer's memory.

Integers are used for counting, which actually happens quite a lot in programming!

```
a = 42 # a is an integer
b = -5 # b is a negative integer
for i in range(0,a): # count i from 0 to a-1
    print(i,"<",a)
```

```
0 < 42
1 < 42
2 < 42
3 < 42
4 < 42
```

This long output was truncated.

```
5 < 42
...
39 < 42
40 < 42
41 < 42
```

L3.3 Floating point numbers

By contrast, floating point numbers aim to represent real numbers (numbers in $\mathbb{R}$): 2.3, $\frac{1}{3}$, $\sqrt{2}$, π... Unfortunately, a computer does not have the infinite precision of math, so always keep in mind the two following principles:

- Any real number will be approximately stored
- Even numbers that seem easy to write, such as 0.1 may be approximated.

The exact reason behind that would be too long to explain here. Let us however give an explanation of the name *floating point*.

To be able to store very small or very large numbers, computers use a version of the *scientific notation*:

- Instead of writing 0.000000000042, write $4.2 \cdot 10^{-11}$
- Instead of writing 35784126.17, write $3.578412617 \cdot 10^{7}$.

In both examples above, the decimal point was moved to have a number between 1 (included) and 10 (excluded); the number of times the decimal point was moved to the left being the exponent (so it is negative when moving to the right). These movements of the decimal point are what is denoted by *floating*.

```
x = 12.0     # x is a float (despite being a whole number)
y = 0.1      # y is imperfectly stored
z = 123.625  # z is perfectly stored
print(x,y,z)
print("%.50f"%x) # Printing with 50 digits after the decimal point
print("%.50f"%y) # Printing with 50 digits after the decimal point
print("%.50f"%z) # Printing with 50 digits after the decimal point
```

```
12.0 0.1 123.625
12.00000000000000000000000000000000000000000000000000
0.10000000000000000555111512312578270211815834045410
123.62500000000000000000000000000000000000000000000000
```

L3.4 Operations on numbers

The usual operations can be performed on both integers and floats. The only operation that needs attention is division: are we using integral division with quotient and remainder (*Euclidean division*), or *real division* (as in division in the real numbers)?

L3.4.1 Addition, subtraction, multiplication

With little surprise, the usual symbols are used:

- `+` for addition
- `-` for subtraction
- `*` for multiplication

```
i = 3
j = 5
a = i+7
b = j-2
c = a*-3
d = i+j
e = b*d
print(a,b,c,d,e)

x = 2.3
y = -5.2
t = x+8.1
u = y-7.1
v = t*2.4
w = x+u
z = y*w
print(t,u,v,w,z)
```

```
10 3 -30 8 24
10.399999999999999 -12.3 24.959999999999997 -10.0 52.0
```

L3.4.2 Division

Real division, meaning the division of two numbers (integers or floats) whose result is a real number, is done with the operator `/`. It always produces floating point numbers, even when the operands are integers and even when the result could be an integer.

```
x = 2/3
y = 16/2
z = 3.5/0.5
# t = 8.6/0 # Don't divide by 0!
print(x,y,z)
```

```
0.6666666666666666 8.0 7.0
```

The *Euclidean division* of positive integers a by $b > 0$ produces two results: the *quotient* q and the *remainder* r. Mathematically, they are defined as the numbers such that $a = b \times q + r$ with $0 \leq r < b$.

This is extended to negative numbers with quotient being negative if the signs of a and b differ and the remainder being $b < r \leq 0$ if $b < 0$.

Note that as in real division you cannot divide by 0.

Since there are two results, there are actually two operators: `//` for the *truncated division* (think quotient) and `%` for the remainder.

```
a = 56
b = -36
c = 7
d = -5
print(a//c,a%c,a//d,a%d,b//c,b%c,b//d,b%d)
edf = "%d = %d × %d + %d" # Format to print the actual equation
print(edf%(a,c,a//c,a%c))
print(edf%(a,d,a//d,a%d))
print(edf%(b,c,b//c,b%c))
print(edf%(b,d,b//d,b%d))
#print(c//0)  # Don't divide by 0!
```

```
8 0 -12 -4 -6 6 7 -1
56 = 7 × 8 + 0
56 = -5 × -12 + -4
-36 = 7 × -6 + 6
-36 = -5 × 7 + -1
```

This also works with floats, but producing float remainders and quotients (even though the quotient is always a whole number.)

```
x = 3.2//2.5
y = 5//1.5
z = 3.5%4
t = 7%2.25
print(x,y,z,t)
```

```
1.0 3.0 3.5 0.25
```

L3.4.3 Conversion between int and float

Conversion can be performed through the `int()` and `float()` functions. The conversion from float to integer *truncates* the number, i.e. just removes the fractional part.

```
a = int(3.85)
b = int(-6.4)
c = int(-8.9)
d = int(12.3)
print(a,b,c,d)
x = float(34)
y = float(-7)
print(x,y)
```

```
3 -6 -8 12
34.0 -7.0
```

L3.4.4 Mixing integers and floats

Mixing integers and floats is possible, but the result will end up being a float. Consequently, conversion from int to float is actually rarely used.

```
x = 3-0.75
y = 2*1.5
z = 7+1.0
print(x,y,z)
```

```
2.25 3.0 8.0
```

L3.4.5 Other operations

Exponentiation is part of the basic operations in Python, and is performed using the ** operator. It works for both integers and floats.

When the exponent is negative, the result is a float even though the operands are integers, because negative exponent means the reciprocal is taken.

And when a negative number is taken to a non-whole exponent, complex numbers are produced (complex numbers will not be covered here).

```
a = 2**6
b = 3**4
c = (-5)**3
d = 2**(-5) # 2**(-5) = 1/(2**5)
print(a,b,c,d)
x = 2.5**3
y = 3**0.5        # Square root of 3
z = (-2)**(0.5)   # Square root of -2: a complex number [note: sqrt(-1) is␣
↪written j instead of i in python]
t = (-0.1)**(-2) # The approximation in 0.1 will show
print(x,y,z,t)
```

```
64 81 -125 0.03125
15.625 1.7320508075688772 (8.659560562354934e-17+1.4142135623730951j)
99.99999999999999
```

L3.4.6 Precedence (a.k.a PEMDAS)

As in mathematics, some operations have *precedence* over others, meaning they must be performed before others. The usual example being $3+2\times 4 = 3+8 = 11$ because multiplication has precedence over addition. To bypass the usual precedence and indicate that an operation must be performed before, parenthesis are used: $(3+2)\times 4 = 5\times 4 = 20$.

The PEMDAS acronym summarizes these rules. It expands in *Parenthesis Exponents Multiplication Division Addition Subtraction*.

It is a useful acronym, although it does not give the full picture. The first thing that is missing and that becomes relevant when writing numbers on a computer is that the negative sign (also known as *unary minus*) actually has more precedence than multiplication and division (but less than exponentiation).

The second thing that is not shown in the accronym is that multiplication and division have **the same** precedence. So in that case the order of writing (*left-to-right associativity*) is used. Similarly, addition and subtraction have **the same** precedence and the order of writing is used to perform the operations.

As a result, a better acronym would be PEN[MD][AS]; not as catchy, but truer!

When using several exponentiation, which by definition all have the same precedence, the operations are performed from *right-to-left*, so using the *reverse* of the order of writing; this is called *right-to-left associativity*. This is in keeping with what happens in mathematics: $2^{3^2} = 2^{\left(3^2\right)} = 2^9 = 512$, while $\left(2^3\right)^2 = 8^2 = 64$.

Also remark that here *division* means both real and Euclidean division.

```
a1 = -3//2      # negative sign has precedence over division
a2 = -(3//2)
b1 = -3**2      # negative sign does not have precedence over␣
 ↪exponentiation
b2 = (-3)**2
c1 = 12//3*4    # division is performed first because it appears first
c2 = 12//(3*4)
d1 = 2-3+5      # subtraction is performed first because it appears first
d2 = 2-(3+5)
e1 = 15//2/4
e2 = 15//(2/4)
f1 = 2**3**2    # exponentiations are performed right to left
f2 = (2**3)**2
print(a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2)
```

```
-2 -1 -9 9 16 1 4 -6 1.75 30.0 512 64
```

L3.4.7 Abbreviated operators

It is often the case that we want to add a value to a variable. One way to do that is to simply have the sum of the variable and the value as the right handside of an assignment:

```
v = 42
print(v)
v = v + 37 # Add 37 to v
print(v)
```

```
42
79
```

Since this kind of operation is very frequent, there is a shortcut additional syntax: `+=`

```
n = 42
print(n)
n += 37 # Add 37 to n; same as n = n + 42
print(n)
```

```
42
79
```

This shortcut syntax is available for all operators:

```
p = 42
print("Original:", p)
p **= 2   # p = p**2
print("After 'p **= 2':",p)
p //= 5   # p = p // 5 (quotient)
print("After 'p //= 5':",p)
p += 27   # p = p + 27
print("After 'p += 27':",p)
p *= 4    # p = p * 4
print("After 'p *= 4':",p)
p -= 1234 # p = p - 1234
print("After 'p -= 1234':",p)
p /= 11   # p = p /11 (real division)
print("After 'p /= 11':",p)
```

```
Original: 42
After 'p **= 2': 1764
After 'p //= 5': 352
After 'p += 27': 379
After 'p *= 4': 1516
After 'p -= 1234': 282
After 'p /= 11': 25.636363636363637
```

Lecture 4
Strings

Lecture contents

L4.1 Introduction **23**
L4.2 Writing strings in the code **23**
L4.2.1 The different kinds of strings 23
L4.2.2 Escaping characters 24
L4.3 Operations on strings **25**
L4.3.1 Concatenation 25
L4.3.2 Repetition 26
L4.3.3 Conversion with other types 27
L4.3.4 On the dangers of loosely typed variables 28
L4.4 Formatted strings **28**
L4.4.1 "Old format" 28
L4.4.2 The `.format()` method 30
L4.4.3 F-strings 32
L4.5 Special characters **33**
L4.5.1 The case of the backspace 33
L4.5.2 Unicode characters 33

L4.1 Introduction

In programming languages, a **string**, or more formally a **string of characters** is the name given to textual data. Strings can be written directly in the program (a.k.a. *verbatim strings*). Another way to create strings is through an input operation.

L4.2 Writing strings in the code

L4.2.1 The different kinds of strings

There are several ways to write verbatim strings, for example to be stored in a variable:

- Single-quoted strings: the text is encased between `'` and `'`.
- Double-quoted strings: the text is encased between `"` and `"`.
- Triple-quoted strings with single quotes: the text is encased between `'''` and `'''`.
- Triple-quoted strings with double quotes: the text is encased between `"""` and `"""`.

```
sqStr = 'Hello!'
dqStr = "How are you?"
tsqStr = '''Good bye!'''
```

```
tdqStr = """It was nice talking."""
print(sqStr,dqStr,tsqStr,tdqStr)
```

```
Hello! How are you? Good bye! It was nice talking.
```

The difference between the several ways of quoting strings lie in the way quotation marks and new lines are handled. Namely, in a string using single quotes, the single quote character cannot be used directly: it would be confused for the end of the string. Similarly, the double-quote character cannot be used directly in double-quoted strings. Triple-quoted version have the same issue, but for three times the character, which is less frequent.

```
sqProblem = 'It's wonderful!'   # Error: the string ends afer It
dqProblem = "He said "Hello!""  # Error: the string ends after said␣
↪(including the space)
tsqProblem = '''The seldom encountered '''triple-quoted''' string in a␣
↪string''' # Error: the string ends after encountered (including the␣
↪space)
```

```
  File "<ipython-input-2-2f058b069eb6>", line 1
    sqProblem = 'It's wonderful!'   # Error: the string ends afer It
                    ^
SyntaxError: invalid syntax
```

L4.2.2 Escaping characters

To be able to enter quotation marks inside a string, they must be *escaped* by preceding them with a backslash: `\'` is a quotation mark, `\"` is a double-quote character.

```
sqEscaped = 'It\'s wonderful!'
dqEscaped = "He said \"Hello!\""
print(sqEscaped,dqEscaped)
```

```
It's wonderful! He said "Hello!"
```

Note that a single quote doesn't have to be escaped in a double-quoted string (though it may be) and vice versa.

```
sqEscaped2 = 'He said "It\'s wonderful\"'          # First " is not␣
↪escaped, second one is
dqEscaped2 = "Is it \"It\'s not\" or \"It isn't\"?" # First ' is escaped,␣
↪second one is not
print(sqEscaped2,dqEscaped2)
```

```
He said "It's wonderful" Is it "It's not" or "It isn't"?
```

Escaped characters allows to write special characters. They all start with a backslash, so the backslash itself now has a special meaning: it also has to be escaped (the list is non-exhaustive):

- `\'`: Single quote (apostrophy)
- `\"`: Double quote
- `\n`: New line
- `\t`: Tabulation
- `\\`: Backslash

An additional (and arguably the principal) difference between the triple-quoted version of strings is that they allow newlines and tabulations to be entered directly, without being escaped. This is why these are sometimes referred to as *multiline strings*.

```
specialStr = "Hi,\nI am writing this\tletter\\message to you." # Escaped␣
↪newline, tabulation, and backslash
specialTqStr = """Hello,
    You see, I don't have to escape tabs\\newlines."""          # Escaped␣
↪backslash only
print(specialStr)
print(specialTqStr)
```

```
Hi,
I am writing this       letter\message to you.
Hello,
    You see, I don't have to escape tabs\newlines.
```

Other than these special characters, the contents of the string are not interpreted. So they can contain about anything, even things that may lool like code or Python special characters:

```
strCode = "Well if you say so #PythonRules" # The 'if' is just a word,␣
↪the # does not start a comment
```

L4.3 Operations on strings

Like other values and variables, strings can be used in operations.

L4.3.1 Concatenation

The `+` operator *adds* two strings in the sense that it puts them together (this is called *concatenation*). The addition between a string and a non-string (say an int or float) results in an error.

```
str1 = "Hello"
str2 = "World"
strConcat = str1+str2         # Adding two strings
print(strConcat)
strConcatError = strConcat+42 # Error: "TypeError: Can't convert 'int'␣
↪object to str implicitly"
```

```
HelloWorld
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-7-a7964ca89db8> in <module>
  3 strConcat = str1+str2         # Adding two strings
  4 print(strConcat)
->5 strConcatError = strConcat+42 # Error: "TypeError: Can't convert 'int'␣
 ↪object to str implicitly"


TypeError: Can't convert 'int' object to str implicitly
```

L4.3.2 Repetition

The * operator *multiplies* a string by repeating it. So it only makes sense to multiply a string by an integer. The multiplication of two strings results in an error.

```
str1 = "Hello "
str2 = "Bye "
strRepeat1 = 3*str1
strRepeat2 = str2*5
strRepeat3 = str1+str2*2 # * has precedence over + (a version of PEMDAS␣
 ↪still applies)
print(strRepeat1)
print(strRepeat2)
print(strRepeat3)
strRepeatError1 = 4.2*"Hi!" # Error: "TypeError: can't multiply sequence␣
 ↪by non-int of type 'float'"
strRepeatError2 = "Ok!"*"Hi!" # Error: "TypeError: can't multiply␣
 ↪sequence by non-int of type 'str'"
```

```
Hello Hello Hello
Bye Bye Bye Bye Bye
Hello Bye Bye

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-9-4ae15ced995b> in <module>
  7 print(strRepeat2)
  8 print(strRepeat3)
->9 strRepeatError1 = 4.2*"Hi!" # Error: "TypeError: can't multiply␣
 ↪sequence by non-int of type 'float'"
 10 strRepeatError2 = "Ok!"*"Hi!" # Error: "TypeError: can't multiply␣
 ↪sequence by non-int of type 'str'"
```

```
TypeError: can't multiply sequence by non-int of type 'float'
```

L4.3.3 Conversion with other types

Other types, in particular integers and floating point numbers can be converted into strings. This is particularly useful when they need to be concatenated with a string.

This is performed by the `str()` function.

```python
val = 42*3+7
strResponse = "The answer is: " + str(val) + "."
print(strResponse)
```

```
The answer is: 133.
```

Conversion from string to integers or floats is possible: the corresponding functions are `int()` and `float()`. It must be noted, however, that these conversions will raise a runtime error if the string does not actually represent a number in a Python acceptable representation. For this to work, the value must be alone in the string; the only thing allowed are spaces before or after. In the case of integers, there can be no decimal point.

```python
strIntValue = "42"            # This is a string, not an int!
intFromStr = int(strIntValue) # OK
strIntValueSpaces = "   35\t"                  # Spaces only are OK
intFromStrWithSpaces = int(strIntValueSpaces) # OK
strInvalidInt1 = "9.0"   # Decimal point
#failedIntFromStr1 = int(strInvalidInt1) # Error: "ValueError: invalid literal for int() with base 10: '9.0'"
strInvalidInt2 = "12in" # Something other than a number
#failedIntFromStr2 = int(strInvalidInt2) # Error: "ValueError: invalid literal for int() with base 10: '12in'"
strInvalidInt3 = "5 3"   # Two numbers
#failedIntFromStr3 = int(strInvalidInt3) # Error: "ValueError: invalid literal for int() with base 10: '5 3'"
strInvalidInt4 = "Hi!"   # No numbers
#failedIntFromStr4 = int(strInvalidInt4) # Error: "ValueError: invalid literal for int() with base 10: 'Hi!'"
```

```python
strIntValue = "42"                # This is a string, not an int!
floatFromStr = float(strIntValue) # OK, is 42.00
strFloatValue = "12.34"           # This is a string, not a float!
floatFromStr = float(strIntValue) # OK, is 42.00
strFloatValueSpaces = "\n75.68"                # Spaces only are OK
intFromStrWithSpaces = float(strIntValueSpaces) # OK
strInvalidFloat1 = "6.54cm"   # Something other than a number
#failedIntFromStr1 = float(strInvalidFloat1) # Error: "ValueError: could not convert string to float: '6.54cm'
strInvalidFloat2 = "5.1 7.3"   # Two numbers
```

```
#failedIntFromStr2 = float(strInvalidFloat2) # Error: "ValueError: could␣
↪not convert string to float: '5.1 7.3'"
strInvalidFloat3 = "Bye..."    # No numbers
#failedIntFromStr3 = float(strInvalidFloat3) # Error: "ValueError: could␣
↪not convert string to float: 'Bye...'"
```

L4.3.4 On the dangers of loosely typed variables

Because the + and * operators can be used for both numbers and strings, albeit with a different meaning (this is called *overloading*), some weird things can happen if the type of variables is unclear. For example, compare the two pieces of code below, where lines 3 and 4 are identical:

```
# Version 1
var = 42  # It's an integer
expr = 4*var
print(expr)
```

```
168
```

```
# Version 2
var = "42" # It's a string
expr = 4*var
print(expr)
```

```
42424242
```

L4.4 Formatted strings

Conversion of numerical values into strings to be concatenated is a frequent thing that is done to output values with a label, for example when debugging or output.

```
x = 42.78
ans = "x="+str(x)
print(ans)
```

```
x=42.78
```

There is an easier way to perform this, which is *formatted strings*. The idea of formatted strings is a string with placeholders, to be filled by values given afterwards.

Several ways to enter formatted strings coexist in Python.

L4.4.1 "Old format"

This way of using formatted strings is sometimes referred to as `printf`*-style* because it uses a similar syntax as is used by the `printf` function in C language.

In the *old format*, placeholders are marked with a percent followed by a letter, to indicate the kind of value:

- `%d` for an integer
- `%f` for a float
- `%s` for a string

Some styling can also be done through the format. For example:

- `%03d` will print an integer with at least 3 digits, the missing ones being replaced by zeroes
- `%.5f` will print a float with 5 digits after the decimal point

For more examples of formats refer to https://pyformat.info/.

```
y = 74
f1 = "The value is %d" # It's a format (just a string, %d is not␣
 ↪interpreted in any way here)
print(f1)
s1 = f1 % y # Replaces %d with 74 in format f1
print(s1)
f2 = "The answer is %03d" # Integer with at least 3 digits (padding with␣
 ↪zeroes)
s2a = f2 % y
s2b = f2 % 147852
print(s2a)
print(s2b)
f3 = "The number is %.5f" # Float with 5 digits after the decimal point
s3a = f3 % (y/3)
s3b = f3 % 1.2
print(s3a)
print(s3b)
```

```
The value is %d
The value is 74
The answer is 074
The answer is 147852
The number is 24.66667
The number is 1.20000
```

Several placeholders can appear in the formatted string. In such cases, there must be a one-to-one correspondence between placeholders and values, which are gathered in a tuple.

```
year = 2000
introduction = "My name is %s, I am %d years old." % ("John Doe",␣
 ↪2020-year)
print(introduction)
```

```
My name is John Doe, I am 20 years old.
```

To use the percent character `%` verbatimly in a formatted string, it has to be released from its placeholder role, and be written `%%` as in the example below.

```
promotion = "The rebate is %f%%." % (100/3) # %f is for the float, %% is %
print(promotion)
```

```
The rebate is 33.333333%.
```

L4.4.2 The `.format()` method

The second way to use formatted strings is through the `.format()` method that can be applied to every string. In this case, placeholders are marked with `{}` regardless of their type (which is the most basic use of the feature). The values corresponding to the `{}`-style placeholders are given as arguments of the `format` function.

```
fName = "John"
lName = "Doe"
age = 42
s = "My name is {} {}, I am {} year old.".format(fName,lName,age)
print(s)
```

```
My name is John Doe, I am 42 year old.
```

But the `format` method offers further advantages. The main one being the possibility to refer to arguments explicitly based on their position in the argument list, or to specify a name for them. It allows to disregard the order of appearance of placeholders, or even to repeat a value.

Positional arguments are specified by integers starting from 0. Named arguments, which must come after any positional arguments, are specified using the syntax `<name>=<value>` in the arguments list.

```
fName = "John"
lName = "Doe"
age = 42
s1 = "My name is {1}, {0} {1}, I am {2} year old.".format(fName,lName,age)
print(s1)
s2 = "My name is {last}, {first} {last}. My first name is {first}, it was␣
 ↪given to me {0} years ago".format(age,first=fName,last=lName)
print(s2)
```

```
My name is Doe, John Doe, I am 42 year old.
My name is Doe, John Doe. My first name is John, it was given to me 42␣
 ↪years ago
```

Styling can also be performed; in that case the type of argument must be specified. The argument, whether implicit, positional, or named, is separated from the format by a colon (:).

```
age=42
sAge1 = "I am {:03d} year old. One year is {:.8f} of my life.".
 ↪format(age,1/age)
```

```
sAge2 = "One year is {1:.8f} of my life because I am {0:+07d} year old.".
↪format(age,1/age)
sAge3 = "One year is {reciprocal:08.4f} of my life because I am {age:
↪=+5d} year old.".format(age=age,reciprocal=1/age)
print(sAge1)
print(sAge2)
print(sAge3)
name = "Albus Percival Wulfric Brian Dumbledore"
sIntro = "My name is {0}, but you can call me {0:^15.9s} for short.".
↪format(name) # Centered in 15 characters, truncated to 9 characters
print(sIntro)
```

```
I am 042 year old. One year is 0.02380952 of my life.
One year is 0.02380952 of my life because I am +000042 year old.
One year is 000.0238 of my life because I am +   42 year old.
My name is Albus Percival Wulfric Brian Dumbledore, but you can call me   ␣
↪Albus Per    for short.
```

Since curly braces are used to denote the placeholders in the format, they have to be doubled in order to appear as curly braces. On the other hand a percent sign no longer serves placeholders and can be used without escaping (i.e. % instead of %%.)

```
val = 42
sSet = "Let N be the set {{0, 1, ..., {}}}.".format(val)
sPercent = "You'll get a {}% rebate".format(val)
print(sSet,sPercent)
```

```
Let N be the set {0, 1, ..., 42}. You'll get a 42% rebate
```

Lists, dictionaries, and tuples (see Lecture L7) can also be used as a named argument. Then the actual index or key is accessed as specified in the format. Note that in dictionaries the key is not enclosed in quotation marks inside the format.

```
jondo = {'first':"John",'last':"Doe",'age':42}
sDict = "My name is {p[first]} {p[last]}, I am {p[age]} year old.".
↪format(p=jondo)
dwarvesL = ["Dopey", "Doc", "Bashful", "Happy", "Grumpy", "Sleepy",␣
↪"Sneezy"]
sList = "Snow White met some of the dwarves: {l[2]}, {l[6]}, {l[0]}".
↪format(l=dwarvesL)
dwarvesT = ("Dopey", "Doc", "Bashful", "Happy", "Grumpy", "Sleepy",␣
↪"Sneezy")
sTuple = "Snow White met some of the dwarves: {t[1]}, {t[3]}, {t[5]}".
↪format(t=dwarvesT)
print(sDict)
print(sList)
print(sTuple)
```

```
My name is John Doe, I am 42 year old.
Snow White met some of the dwarves: Bashful, Sneezy, Dopey
Snow White met some of the dwarves: Doc, Happy, Sleepy
```

For more example of formats refer to https://pyformat.info/.

L4.4.3 F-strings

Remark on Python's version

This method only works with Python 3.6 and over. Older versions of Python will produce `SyntaxErrors` when trying to run the code below. You can check the version that is currently running as follows:

```
import sys
print("The current Python version is",sys.version)
```

```
The current Python version is 3.7.6 (default, Feb  3 2020, 16:05:52)
[GCC 7.4.0]
```

F-strings are a more readable version of formatted strings using the `.format()` method. It works as follows:

- The string is preceded by the letter `f`: `f"My String"` is an f-string (`F` also works)
- Placeholders are between curly braces; they use the format `{<value>:<style>}`. The styling is the same as in the `.format()` version. The value can be anything that can be evaluated: calculation, function call, etc.

```
def myfun(n):
    return n-17
age=42
s1 = f"I am {age} year old."
s2 = f"I am level {age**2:08d}."
s3 = f"I tell everyone I am {myfun(age)} year old."
print(s1,s2,s3)
```

```
I am 42 year old. I am level 00001764. I tell everyone I am 25 year old.
```

When using values from dictionaries, quotation marks must be used, because what is inside the curly braces is just normal Python code. That is when using different quotation marks comes in handy.

```
jondo = {'first':"John",'last':"Doe",'age':42}
fsDict = f"My name is {jondo['first']} {jondo['last']}, I am␣
↪{jondo['age']} year old."
print(fsDict)
```

```
My name is John Doe, I am 42 year old.
```

One additional caveat is that f-strings cannot include a backslash inside the curly braces, not even within a string.

```
fsBslashOk = f"I just want to say:\n\t{'Hello'}" # In the string part: OK
print(fsBslashOk)
#fsBslashKo = f"You replied {'Good\nBye'}" # In the expression part: Error
#print(fsBslashKo)
```

```
I just want to say:
        Hello
```

L4.5 Special characters

L4.5.1 The case of the backspace

The *backspace* character, escaped as `\b` is used to delete content. But it actually behaves a bit differently in a terminal and in Jupyter:

- In a terminal, the backspace character moves the position of the writing cursor one character to the left. It does not actually erase anything.
- In Jupyter, it actually erases, but there are bugs when several are printed together, so they have to be printed one by one, and the print should flush the output.

```
print("Hello","\b",sep="")
print("Goodbye","\b"*5,"|",sep="")             # In Jupyter: prints
 ↪"Goodby|". In Terminal: prints "Go|dbye"
print("Goodbye","\b"*5," "*5,"\b"*5,"|",sep="") # In Jupyter: prints
 ↪"Goodby     |". In Terminal: prints "Go|"
print("Goodbye",end="")
print("\b",end="",flush=True)
print("\b",end="",flush=True)
print("\b",end="",flush=True)
print("\b",end="",flush=True)
print("\b",end="",flush=True)
print("|")                                      # In Jupyter: prints
 ↪"Go|".  In Terminal: prints "Go|dbye"
```

L4.5.2 Unicode characters

Among special characters are symbols, such as accented letters or non-latin characters, or even emojis. Recent versions of Python support unicode characters by default, but they are not always easy to write in the code.

Also note that the symbol must be supported by the reader (browser, terminal, etc) in order to be actually displayed.

```
symb = "Let α=2×x^3"
print(symb)
```

```
Let α=2×x^3
```

A way to write them is to use their hexadecimal code directly. For symbols with a code on only 4 hex-digits, the 16-bit version can be used: `\uXXXX` where `XXXX` is the code. When the 8 hex-digit (32-bit) version is needed, use `\Uxxxxxxxx`. In both cases you must specify the 4 or 8 hex-digits, using zeroes if need be.

The code for a specific character can be found on the Unicode Consortium's website: https://home.unicode.org/.

```python
learnDingbats = "Learn Python \u2714" # "Checkmark" dingbat
ilPython = "I \U0001F9E0 Python"       # Brain emoji
ilnyDingbats = "I \U00002764 NY"       # "Heart" dingbat/emoji, could also be coded as \u2764
ilnyEmojis = "I \U0001FAC0 NY"         # "Anatomical heart" emoji - New addition in 2020, may not show
alphabet = "\u0061\u0062\u0063\u0064" # Normal letters have a code too, but that is not very efficient
god = "\u2728\U0001F4DC\u2728 I am the \u03B1 and the \u03C9" # Emojis and greek letters
print(learnDingbats,"-",ilPython)
print(ilnyDingbats,"-",ilnyEmojis)
print(alphabet)
print(god)
```

```
Learn Python ✓ - I 🧠 Python
I ♡ NY - I 🫀 NY
abcd
✨📜✨ I am the α and the ω
```

Lecture 5
Conditions

Lecture contents

L5.1 Introduction **35**
L5.2 Writing conditions **35**
L5.2.1 Boolean constants 36
L5.2.2 Arithmetic expressions 36
L5.2.3 Comparison operators 36
L5.2.4 Boolean operators 38
L5.3 Conditional structure: `if`/`else` **40**
L5.3.1 Example: Positive or Not positive 40
L5.3.2 Example : Pass or Fail 41
L5.3.3 Example: Parity 41
L5.4 Multiple cases: `if`/`elif`/`else` **41**
L5.4.1 Example: Positive, Zero, Negative 42
L5.4.2 Example: Rock, Paper, Scissors 42

L5.1 Introduction

One of the main ability of computer programs is to react differently based on the data it manipulates, usually a user's input.

For example, when buying a taxable product in New York City, an extra 8.875% of the price is added as a sales tax. But in Boston, the sales tax is only 6.25% of the price. So the total price of a taxable product depends on the location.

These choices are performed using the `if` keyword in Python.

```
if condition: # Do not forget the colon (:)!
    something # Do not forget the indentation!
```

When the condition is true, the `something` part is executed. Otherwise it is skipped.

L5.2 Writing conditions

The `condition`, is an expression which evaluates to either `True` or `False`. It is a called as a *Boolean expression*. Simple Boolean expressions can be combined using *conditional operators*.

L5.2.1 Boolean constants

The simplest condition that can be written are the constants `True` and `False`. Remark that they start with an uppercase letter.

Note that using these as a condition is not very interesting because:

```
if True:
    something
```

is equivalent to simply writing `something`

and

```
if False:
    something
```

is equivalent to writing nothing.

L5.2.2 Arithmetic expressions

Arithmetic expressions also carry a truth value: any non-zero value means `True` and zero means `False`. Using arithmetic expressions as conditions confuses integers (or floats) with boolean values, and should be avoided if possible.

L5.2.3 Comparison operators

One way to write an expression that evaluates to `True` or `False` is by using a *comparison operator*.

The comparison oerator for equality is `==`: when `a` is equal to `b`, then the value of `a==b` is `True`. Otherwise, it is `False`.

Example: Is the input 5?

The following code uses `==` to test whether the input `x` is equal to `5`. In that case only it prints `5 is five`.

```
x = int(input('Type an integer x: '))
if x==5:
    print(x,'is five.') # Print the value of x followed by 'is five'
```

```
Type an integer x: 5
5 is five.
```

== vs =

The *assignment operator* `=` must not be confused with the *equality operator* `==`. It is a common mistake and a frequent typo. In Python, using `=` in a conditional produces a syntax error so it is easily catched (this is not always the case in other languages).

```
x = 1        # Assignment
x == 2       # Equality test (not printed), no assignment performed
print("x =",x)
y = 2
z = 1
print(x==y)
print(x==z)
```

```
x = 1
False
True
```

```
if x=y: # Typo that creates a syntax error
    print('x equals y')
```

```
      File "<ipython-input-10-d409dd90c269>", line 1
    if x=y: # Typo that creates a syntax error
        ^
SyntaxError: invalid syntax
```

Other comparison operators

Comparisons are not restricted to equality. Here is the complete list of comparison operators:

- `==`: equal
- `!=`: not equal
- `<`: less
- `<=`: less or equal to
- `>`: greater
- `>=`: greater or equal to

```
x = 1 # Assignment
print(x>2, x<=1, x==1, x!=2, x>=4.3, x<3)
```

```
False True True True False True
```

In the above code, first `x = 1` assigns `1` to `x`. Then:

- `x>2` evaluates the truth value of the statement $1 > 2$, which is `False`.
- `x<=1` evaluates the truth value of the statement $1 \leq 2$ which is `True`.
- `x==1` evaluates the truth value of the statement $1 = 1$ which is `True`.
- `x!=2` evaluates the truth value of the statement $1 \neq 2$ which is `True`.
- `x>=4.3` evaluates the truth value of the statement $1 \geq 4.3$ which is `False`.
- `x<3` evaluates the truth value of the statement $1 < 3$ which is `True`.

Comparison of string objects

Strings can also be compared to one another. Equality and non-equality work as expected (they are case sensitive), but inequality operators use the *lexicographic order* (the order of the dictionary) with uppercase letters before lowercase letters, so they should be used with care.

```
s1 = "Hello"
s2 = "hello"
s3 = "hello"
s4 = "hell"
s5 = "hi"
```

```
s6 = "a"
s7 = "Z"
s8 = "z"
print(s1,"==",s2,":",s1==s2)
print(s1,">=",s2,":",s1>=s2)
print(s2,">",s1,":",s2>s1)
print(s2,"==",s3,":",s2==s3)
print(s2,"<",s4,":",s2<s4)
print(s2,"<",s5,":",s2<s5)
print(s6,">",s7,":",s6>s7)
print(s6,">",s8,":",s6>s8)
```

```
Hello == hello : False
Hello >= hello : False
hello > Hello : True
hello == hello : True
hello < hell : False
hello < hi : True
a > Z : True
a > z : False
```

Comparisons of strings for equality or inequality is however widespread:

```
word1 = input('Type something: ')
word2 = input('Type something else: ')
if word1==word2:
    print("You wrote the same thing twice")
```

```
Type something: Hello
Type something else: Hello
You wrote the same thing twice
```

L5.2.4 Boolean operators

Boolean expressions such as the results of comparisons can be connected through Boolean operators to form compound conditions.

There are three main boolean operators:

- `or`: Boolean OR (disjunction): *true* when **at least one** operand is *true*
- `and`: Boolean AND (conjunction): *true* whan **all** the operands are *true*
- `not`: Boolean NOT (negation): changes *true* in *false* and *false* in *true*

Example: Compound Boolean expressions

```
x = 1
print(not x>2, x>0 and x<1, x>0 or x<1, -2<x<=2, -2<x and x<=2) # The
 ↪last two are equivalent
```

```
True False True True True
```

P	Q	P or Q
T	T	T
T	F	T
F	T	T
F	F	F

(a) Truth table for the *or* operator

P	Q	P and Q
T	T	T
T	F	F
F	T	F
F	F	F

(b) Truth table for the *and* operator

P	not P
T	F
F	T

(c) Truth table for the *not* operator

Table L5.1: Truth tables for the *or*, *and*, and `not` Boolean operators.

First, `x = 1` assigns `1` to `x`. Then:

- `not` operators reverses the result. So `not x>2`, first evaluates the truth value of $1 > 2$, which is `False`, then evaluates `not False`, which is `True`.
- $1 > 0$ is `True` but $1 < 1$ is `False`.
  - For `x>0 and x<1`, it evaluates the value of `True and False`, which is `False` since the `and` operator gives `True` only if both operands are `True`. Therefore `x>0 and x<1` is `False`.
  - For `x>0 or x<1`, it evaluates the value of `True or False`, which is `True` since the `or` operator gives `True` when at least one operand is `True`. Therefore `x>0 or x<1` is `True`.
- `-2<x<=2` is a shorthand for `-2<x and x<=2`. $-2 < 1$ and $1 \leq 2$ both evaluate to `True`, so the conjunction is `True`.

Truth tables

The tables of Table L5.1, called *Truth tables*, give the truth value (true, T, or false, F) of the compound condition based on the truth value of the operands.

`P or Q` is `False` only when `P=False` and `Q=False`. Note that when both `P` and `Q` are true, in computers `P or Q` is true. It is usually not the case in English where "chicken or fish" does not mean that "both" is an option.

`P and Q` is `False` only when `P` is `True` and `Q` is `False`.

`not P` is `True` when `P` is `False`.

Truth tables for complex expressions Using `or`, `and`, and `not`, complex Boolean expressions can be written. When these are true depending on `P` and `Q` can be calculated using truth tables: calculate the truth value of each sub-expression in a column, combining columns using the truth table for the operators.

So based on Table L5.2(a), `(not P) or Q` is `False` only when `P` is `True` but `Q` is `False`.

De Morgan's Laws Comparing the truth tables for `(not P) and (not Q)` in Table L5.2(b) and `not(P or Q)` in Table L5.2(c) shows that they have the same value for any combination of truth values of `P` and `Q`: they actually represent the same expression. This particular equality is one of the two De Morgan's laws which can be stated as:

- `(not P) and (not Q)` is equivalent to `not(P or Q)`.
- `(not P) or (not Q)` is equivalent to `not(P and Q)`.

P	Q	not P	(not P) or Q
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

(a) Truth table for `(not P) or Q`

P	Q	not P	not Q	(not P) and (not Q)
T	T	F	F	F
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

(b) Truth table for `(not P) or Q`

P	Q	P or Q	not(P or Q)
T	T	T	F
T	F	T	F
F	T	T	F
F	F	F	T

(c) Truth table for `(not P) or Q`

Table L5.2: Truth tables for complex expressions.

L5.3 Conditional structure: `if/else`

In order to provide a different result based on a `condtion`, some code should still be executed when the condition evaluates to `False`. This is coded with an `if/else` structure. It is sometimes called *if/then/else*, but Python does not use the *then* keyword. The syntax is as follows:

```
if condition:           # Colon
    something1          # Indentation of the "then" block
else:                   # Colon
    something2          # Indentation of the "else" block
```

When the `condition` is `True`, `something1` is executed. Otherwise (when the `condition` is `False`), run `something2` code.

L5.3.1 Example: Positive or Not positive

To test whether a number x is positive or not, the following procedure (this is English, not Python) can be used:

```
Get an integer x from a user.
If x is positive:
  print the value of x and "is positive."
else:
  print the value of x and "is not positive."
```

This can be converted into Python:

```
x = int(input('Type an integer x: '))
if x>0:
    print(x,"is positive.")
else:
    print(x,"is not positive.")
```

```
Type an integer x: 42
42 is positive.
```

L5.3.2 Example : Pass or Fail

Consider the following grading rule for a test: > If the test score is more than `60`, the grade is `Pass` Otherwise, the grade is `Fail`.

To implement it as a code, first it can be expressed with more details as a procedure:

```
Get a test score as an integer from a user.
If score is greater than 60:
  print 'Pass'
else:
  print 'Fail'
```

It can then be translated into Python:

```
score = int(input('Type your score as an integer: '))
if score>60:
  print('Pass')
else:
  print('Fail')
```

```
Type your score as an integer: 42
Fail
```

L5.3.3 Example: Parity

To check whether a number is even or odd, it is divided by 2. If the remainder is 0, then the number was even, otherwise it was odd:

```
x = int(input('Type an integer x: '))
if x%2==0:
    print(x,"is even")
else:
    print(x,"is odd")
```

```
Type an integer x: 29
29 is odd
```

L5.4 Multiple cases: `if/elif/else`

The `if/else` structures handles only two possibilities. By using `elif` ("else if") multiple times, we can handle multiple cases.

```
if condition1:
    something1
elif condition2:
    something2
```

```
elif condition3:
    something3
....

else:
    something
```

Notes:

- The `else` **cannot** have a condition.
- The `else` is actually optional if the `something` is nothing.

L5.4.1 Example: Positive, Zero, Negative

The example above (Section L5.3.1) did not distinguish the special case $x = 0$. Let's refine it so that:

- if x is positive, the program prints "Positive"
- else if x is zero, the program prints "Zero"
- otherwise, the program prints "Negative"

An `if/elif/else` is used to handle all 3 cases: positive, zero and negative.

```
if  x is positive:
    print "Positive"
elif x is 0:
    print "Zero"
else:
    print "Negative"
```

In Python, it becomes:

```python
x = int(input("Type an integer x: "))
if x>0:
    print("Positive")
elif x==0: # Use == for equality!
    print("Zero")
else:
    print("Negative")
```

```
Type an integer x: 0
Zero
```

L5.4.2 Example: Rock, Paper, Scissors

The following example could be part of a *Rock, Paper, Scissors* game, namely a part dealing with interpreting user input:

- If a user types `r`, print `Rock`.

- If a user types `p`, print `Paper`.
- If a user types `s`, print `Scissors`.
- Otherwise, print nothing.

Remark that here the input is to be treated as a `string` type, so there is no conversion to `int` or `float`. Also, `r` and `p` are letters, not variables. So they must be enclosed in quotation marks like `'r'` and `'p'` in Python.

```
play = input('Please, type r, p or s: ')
play = play.lower() # convert input to lower case.
if play=='r':
    print("Rock")
elif play=='p':
    print("Paper")
elif play=='s':
    print("Scissors")
# No else case needed
```

```
Please, type r, p or s: p
Paper
```

Lecture 6
Loops

Lecture contents

L6.1 Programming Structures for Iteration **45**
L6.2 The `while` loop **45**
L6.2.1 Syntax of `while` 46
L6.2.2 Example: Integral division 46
L6.2.3 The `break` and `continue` keywords 47
L6.2.4 Example: The free-throw challenge 48
L6.3 The `for` loop **49**
L6.3.1 Syntax of `for` 49
L6.3.2 Examples: The `range` function 50
L6.3.3 Example: The sum of the first `n` integers 50
L6.4 Example of loop usage **50**

L6.1 Programming Structures for Iteration

We hardly ever do things once. In general mastering various tasks or skills requires several iterations or repetitions As programming languages are modeled after human behavior, any programming language has programming structures for iterations. In particular, the Python language has two types of iteration:

1. the `while` loop for *indefinite* repetition
2. the `for` loop for *definite* repetition.

L6.2 The `while` loop

The `while` loop is used with indefinite repetition. An iteration is called indefinite if it is not known in advance of the number of iterations. For instance, consider a program that starts by asking the user for a pair of integers where the second number cannot equal 0. As long as the user enters `0` for the second number, the code has to repeat asking the user for a pair. Since it is not known in advance on how many times we need to ask the user for such pair, such repetition is deemed indefinite.

Another example may involve free-throwing a basketball until 3 consecutive score baskets are achieved. Once again, one cannot tell in advance how may free-throws are needed to achieve this goal.

L6.2.1 Syntax of `while`

To code a **`while`** loop, a boolean expression and a body are needed. It has the following format:

```
while condition: # Do not forget the colon
    statement 1
    statement 2  # Body of the loop is indented
    statement 3
    .
    .
    .
    statement n
```

The condition is a Boolean expression: a logical expression which evaluates to either `True` or `False`. It may involve:

- boolean values `True`, `False`;
- relational operators: `<`, `>`, `<=`, `>=`, `==` (is equal), `!=` (is different);
- logical operators: `not`, `and`, `or`;
- arithmetic expressions: non-zero (in most cases) converts to `True`, 0 converts to `False`.

For mode details on conditions see Lecture L5.

The condition ends with colon `:`. The omission of the colon is an error and the execution of the code will be interrupted there.

The body of the loop consists of one or more statements, all having an additional level of indentation from the loop.

When the condition evaluates to `True`, the loop body is executed. Then the code goes back to checking the condition, and so on. When the condition evaluates to `False`, the while loop is finished and the code that follows is executed.

L6.2.2 Example: Integral division

The code below computes the quotient and remainder for the integer division of a pair of integers. Since division by 0 cannot be meaningfully defined, it makes sure that the second integer, the divisor, is different from 0.

```
a = input('Enter an integer a = ')
a = int(a)

failedinput = True
while failedinput:
    b = input('Enter a non-zero integer b = ')
    b = int(b)
    if b != 0:
        failedinput = False

q = a//b   # Integer divison operator (quotient)
r = a%b    # Integer remainder operator
print('The quotient  of {}/{} is {}.'.format(a,b,q))
```

```
print('The remainder of {}/{} is {}.'.format(a,b,r))
print('The long division algorithm: {}={}*{}+{}.'.format(a,b,q,r))
```

```
Enter an integer a = 34
Enter a non-zero integer b = 0
Enter a non-zero integer b = 0
Enter a non-zero integer b = 0
Enter a non-zero integer b = 7
The quotient  of 34/7 is 4.
The remainder of 34/7 is 6.
The long division algorithm: 34=7*4+6.
```

Note: In this example there is also an `if` statement which is a single selection programming structure. The syntax is similar to that of the `while`. The main difference between these two programming structures is that `if` executes its body at most once whereas `while` may execute its body several times.

L6.2.3 The `break` and `continue` keywords

The main way for a `while` loop to stop is when the condition becomes `False`. There is an additonal possibility to *break* the execution of the loop: the `break` keyword. When the `break` keyword is executed, the loop stops execution the subsequent statements in the body and does not try to evaluate the condition anymore; it just moves to the next instruction after the loop.

```
secretNumber = 42
while True: # This can never evaluate to False
    guess = int(input("Guess my secret number: "))
    if guess == secretNumber:
        break
    print("Wrong! Try again.")
print("Congratulations!")
```

```
Guess my secret number: 10
Wrong! Try again.
Guess my secret number: -7
Wrong! Try again.
Guess my secret number: 42
Congratulations!
```

The `continue` keyword skips the rest of the body of the loop, and goes back directly to the beginning of the loop, testing the condition again:

```
x=0
while x<16:
    x += 1
    if x%2==0:
        continue # Skip the rest of the body(the print)
    print("x =",x)
```

```
print("The loop has stopped")
```

```
x = 1
x = 3
x = 5
x = 7
x = 9
x = 11
x = 13
x = 15
The loop has stopped
```

L6.2.4 Example: The free-throw challenge

This example simulates the free-throw challenge. Namely, we want to see how many free-throws are needed to score 3 consecutive baskets. Scoring a basket is simulated through probabilities. Suppose that there is a chance of 1 in 10 to score. Using the random integer function in Python, the program can generate a random number in the range 1 through 10. If the number is 10, it counts as a score, otherwise it counts as a miss.

```
import random

def main():
    print("Let's simulate the 3 score in a row free-throw game")
    tcount = 0                     # Total number of throws
    consecutivecount = 0           # Number of consecutive scores
    while consecutivecount < 3:
        v = random.randint(1,10) # Random value between 1 and 10
        tcount += 1
        if v == 10:
            consecutivecount += 1
            print('Throw {} success: v = {}'.format(tcount,v))
        else:
            print('Throw {} fail: v = {}'.format(tcount,v))
            consecutivecount = 0 # reset counter

    print('Challenge done in {} throws.'.format(tcount))

main()
```

```
Let's simulate the 3 score in a row free-throw game
Throw 1 fail: v = 9
Throw 2 fail: v = 8
Throw 3 fail: v = 6
Throw 4 fail: v = 2
Throw 5 fail: v = 1
Throw 6 fail: v = 7
Throw 7 fail: v = 8
```

```
Throw 8 fail: v = 7
Throw 9 fail: v = 8
Throw 10 fail: v = 7
Throw 11 fail: v = 5
Throw 12 fail: v = 6
Throw 13 fail: v = 2
Throw 14 fail: v = 2
Throw 15 fail: v = 7
Throw 16 fail: v = 7
Throw 17 success: v = 10
...
Throw 2743 fail: v = 6
Throw 2744 success: v = 10
Throw 2745 fail: v = 9
Throw 2746 fail: v = 5
Throw 2747 success: v = 10
Throw 2748 success: v = 10
Throw 2749 success: v = 10
Challenge done in 2749 throws.
```

This really long output was truncated.

Note: Slight adjustments to the above code can be made in case the chances of scoring are higher or lower. As an exercise, think about how the above code should be modified if, say, there the chances of scoring a basket are 1 in 5.

L6.3 The `for` loop

The `for` loop is used with definite iteration or repetition. An iteration is called definite if the number of iterations is known in advance. For instance, consider a program that computes the sum of the first 1000 positive integers. Since it is known in advance how many times the program needs to add, in this case 1000 times, the iteration is definite.

Another example may involve free-throwing a basketball until 3 consecutive score baskets are achieved. But this time, there is a cap say of 100 on the number of throws. Once again, one can tell in advance the maximum number of free-throws, a 100 in this case, so the iteration is definite.

L6.3.1 Syntax of `for`

A `for` loop requires a counter object and a rule for the counter. It has three main formats:

```
for i in range(a): # i runs from 0 to a-1 in increments of 1
    statements
```

```
for i in range(a,b): # i runs from a to b-1 in increments of 1
    statements
```

```
for i in range(a,b,c): # i runs from a to b-1 in increments of c
    statements
```

L6.3.2 Examples: The `range` function

These examples explore the rule for the counter.

```
for i in range(5):
    print(i, end=' ')  # Specify the last character to be printed␣
 ↪(default is newline)
print()                # Print a new line
for i in range(3,7):
    print(i, end=' ')
print()
for i in range(0,100,5):
    print(i, end=' ')
print()
for i in range(100,0,-10):
    print(i, end=' ')
```

```
0 1 2 3 4
3 4 5 6
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
100 90 80 70 60 50 40 30 20 10
```

L6.3.3 Example: The sum of the first `n` integers

This example computes the sum of the first `n` positive integers, for example the sum or `1+2+..+50`, or `1+2+3+..+2020`.

```
def main():
    n = input('Enter a positive integer n = ')
    n = int(n)
    s = 0
    for i in range(1,n+1): # i goes from 1 to n
        s += i
    print('1 + 2 + ... + {} = {}.'.format(n,s))
main()
```

```
Enter a positive integer n = 2020
1 + 2 + ... + 2020 = 2041210.
```

L6.4 Example of loop usage

This example computes the sum and the average value of a list of numbers. The program asks the user to enter numbers or 'quit'. To read these numbers however many as they are, an indefinite `while` loop is used. As the numbers are read, they are stored into a list object. Since the list object knows how many items it has, a definite `for` loop is used to compute the sum and average.

Note: When working with a list we should keep in mind the following:

- lists are intialized using `= []` for empty lists

- the `append()` function is used to insert at the end of the list
- the objects in the list may have different types
- the `len()` function returns the size of the list (the number of objects)
- the bracket operator `[]` with an index is used to access (read/write) the elements; for ex `num[10]` is the 11th element
- items in the list are indexed from `0` to `len(list)-1`

For a deeper look on lists, see Lecture L7.1.

```
def main():
    num = []
    while True:
        n = input('Enter a number or \'quit\' to finish: ')
        if n == 'quit':
            break  # Exit the loop
        n = float(n)
        num.append(n)
    s = 0
    for i in range(len(num)):
        s += num[i]
    av = s/len(num)
    print('The entries of the list are: {}'.format(num))
    print('The sum of the entries is {}.'.format(s))
    print('The average of the entries is {}.'.format(av))
main()
```

```
Enter a number or 'quit' to finish: 2.3
Enter a number or 'quit' to finish: 1.2
Enter a number or 'quit' to finish: 5.4
Enter a number or 'quit' to finish: quit
The entries of the list are: [2.3, 1.2, 5.4]
The sum of the entries is 8.9.
The average of the entries is 2.966666666666667.
```

Lecture 7

Lists, Tuples, Dictionaries

L7.1 Lists

L7.1.1 What is a list?

A variable can store exactly one given value at a time. A **list**, on the other hand, is a special type of variable that can store more than one value:

```
number = 4
numbers = [] # Empty list
numbers = [1,5,7,9,2]
```

The list `numbers` starts with an opening bracket `[` and ends with a closing bracket `]`. The space between the brackets is filled with five numbers separated by commas.

The elements inside a list may have different types. Some of them may be integers, others floats, and yet others may be lists. The following example shows a *heterogeneous* list with elements of different types: integer, float, string and another list

```
heterogeneous_list = [3, 4.5, "hello", [2, 3.0]]
```

L7.1.2 List indexing

The elements in a list are numbered starting from zero. This means the element stored at the beginning of the list has the index number `0` and the last element is assigned the index number `length of the list -1`.

Consider the `numbers` list above:

- The number at position (index) 0 is 1
- The number at index 1 is 5
- The number at index 2 is 7
- The number at index 3 is 9
- The number at index 4 is 2

L7.1.3 Printing elements of a list

Elements of a list can be printed separately by referring each element by its index.

```
print(numbers[0], numbers[1], numbers[2], numbers[3], numbers[4])
```

```
1 5 7 9 2
```

Python can print the entire list only by its name as well.

```
print(numbers)
```

```
[1, 5, 7, 9, 2]
```

L7.1.4 Accessing an element

A specific element in a list is referred by the name of the list followed by the index number in brackets. This is used both to use the value of the element or to change it.

```
numbers[0] = 3          # Assigns 3 to the value of the element at index␣
 ↪0
print(numbers)
numbers[1] = numbers[4] # Assigns the value of the element at index 4 to␣
 ↪the value at index 1
print(numbers)
```

```
[3, 5, 7, 9, 2]
[3, 2, 7, 9, 2]
```

L7.1.5 Length of a list

A list can grow or shrink during the execution of the program: elements can be added or deleted. To find the current number of elements of a list, use the `len()` function with the name of the list as an argument. It returns the number of elements.

```
print(len(numbers))
```

```
5
```

L7.1.6 Negative indices

Python allows negative indices. While positive indices starts from 0 (index position of the first element in the list) and increase as we move towards the end of the list, negative indices starts from -1 (index position of the last element in the list) and decreases as move towards the beginning of the list.

For example:

Positive indices:		0	1	2	3
List:	[	3,	2,	7,	9]
Negative indices:		-4	-3	-2	-1

```
# Both indices refer to the last element
print(numbers[-1])
print(numbers[len(numbers)-1])
```

```
2
2
```

L7.1.7 Removing an element

Any of the list's elements may be removed at any time – this is done with an instruction named *del*. `del` is not a function but an instruction.

```
print(numbers)
del numbers[4] # Delete the element at index 4
print(numbers)
print("Length:", len(numbers))
```

```
[3, 2, 7, 9, 2]
[3, 2, 7, 9]
Length: 4
```

It is impossible to access an element which does not exist, whether to use its value or assign it. The following instruction causes a runtime error: index 4 does not exist anymore since it was in the instruction above.

```
numbers[4] = 1
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-12-52a8b450472a> in <module>
->1 numbers[4] = 1


IndexError: list assignment index out of range
```

L7.1.8 Appending an element to a list

To append an element to a list, a special method (not a function) called `append()` is used. A method looks like a function but it differs in the way how it is invoked. A method is owned by the data it works for. This means that invoking a method requires some specification of the data from which the method is invoked.

In general, a function invocation may look like:

`result = function(arg)`

The function takes an argument, does something, and returns a result.

A typical method invocation usually looks like:

`result = data.method(arg)`

Here is how the value 4 can be added to the list numbers:

```
print('Before:', numbers)
numbers.append(4)
print('After appending 4:', numbers)
```

```
Before: [3, 2, 7, 9]
After appending 4: [3, 2, 7, 9, 4]
```

L7.1.9 Inserting an element into a list

While `append()` method adds a value to the end of a list, `insert()` can add the value at a specific position in the list. It takes two arguments:

`list.insert(where, what)`

The first gives the location of the element to be inserted (index), the second is the element to be inserted. All the existing elements that occupy locations to the right of the new element (including the one at the indicated position) are shifted to the right, in order to make space for the new element

```
print("Before:", numbers)
numbers.insert(0,232)   # Insert 232 at index 0
numbers.insert(-1, 444) # Insert 444 at index -1
print("After:", numbers)
```

```
Before: [3, 2, 7, 9, 4]
After: [232, 3, 2, 7, 9, 444, 4]
```

L7.1.10 Sorting a list

To sort elements in ascending order the `sort()` method is used:

```
list = [8, 10, 6, 2, 4]
list.sort()
print(list)
```

```
[2, 4, 6, 8, 10]
```

L7.1.11 Lists and loops

Example: Creating a list from scratch

A list's life can start as the empty list to which new elements are added as needed with `append()` method:

```
list = []
for i in range(5):
    list.append(i+1)
print(list)
```

```
[1, 2, 3, 4, 5]
```

Another way is to use the insert() method:

```
list = []
for i in range(5):
    list.insert(0, i+1)
print(list)
```

```
[5, 4, 3, 2, 1]
```

Example: Calculating the sum of the elements in a list

A variable `sum` is used to store the sum and initially assigned a value of 0. Then all the elements of the list are added to `sum` using a for loop.

```
list = [ 10, 1, 8, 3, 5]
sum = 0
for i in range(len(list)):
    sum += list[i]
print(sum)
```

27

In the above code, `list` is assigned a sequence of five integer values. The `i` variable takes the values 0, 1, 2, 3, and 4, and then it indexes the list, selecting the subsequent elements: the first, second, third, fourth and fifth. Each of these elements is added together by the `+=` operator to `sum` variable, giving the final result at the end of the loop.

See Section L3.4.7 for more information on the += operator.

Example: Traversing a list using the `in` keyword

A `for` loop can do much more with a list. It can hide all the actions connected to the list's indexing, and deliver all the list's elements. This ability of list is called being a *sequence type*.

```
list = [ 10, 1, 8, 3, 5]
sum = 0
for element in list:
    sum += element
print(sum)
```

27

The `for` instruction specifies the variable used to browse the list (`element` here) followed by the `in` keyword and the name of the list being processed (`list` here). The `element` variable is assigned the values of all the subsequent list's elements, and the process occurs as many times as there are elements in the list. This means that the `element` variable is a copy of the elements' values. Indices and the `len()` function are therefore not needed here.

L7.1.12 Slices

A slice is an element of Python syntax that allows to make a brand new copy of a list, or parts of a list.

The syntax is as follows:

`listname[startindex:endindex]`

It resembles indexing, but the colon inside makes a big difference. A slice of this form makes a new (target) list, taking elements from the source list – the elements of the indices from `startindex` to `endindex-1`. The element at `endindex` is not included in the slice.

Using negative values for both start and end is also possible.

```
list1 = [10,8,6,4,2]
new_list = list1[1:3] # Only elements at index positions 1 and 2 are included in the new_list
```

```
print(new_list)

# Negative index example
list1=[10,8,6,4,2]
new_list= list1[1:-1] # Starting index is 1 ending index is -2 (positive␣
↪equivalent is 3).
print(new_list)
```

```
[8, 6]
[8, 6, 4]
```

The starting index, ending index, or both can be omitted in a slice. If this happens default values are used.

```
list1 = [10, 8, 6, 4, 2]
newlist1 = list1[:]  # Both indices are omitted. So start is 0 and end is␣
↪length of the list -1
print(newlist1)
newlist2 = list1[:3] # Start index is 0.
print(newlist2)
newlist3 = list1[2:] # End index is 5
print(newlist3)
```

```
[10, 8, 6, 4, 2]
[10, 8, 6]
[6, 4, 2]
```

Slices can also be used with `del` instruction to delete a range of elements at once:

```
list1 = [10,8,6,4,2]
del list1[1:3]
print(list1)
```

```
[10, 4, 2]
```

L7.1.13 Searching for an element in a list

Python offers two very powerful operators, able to look through the list in order to check whether a specific value is stored inside the list or not. These operators are `in` and `not in` operators. The syntax is as follows:

```
element in list
element not in list
```

The first of them (`in`) checks if a given element (its left operand) is currently stored somewhere inside the list (the right operand). In this case the operator returns `True`.

The second (`not in`) checks if a given element (its left operand) is absent in a list. In this case the operator returns `True`.

```
list = [0, 3, 12, 8, 2]
print(5 in list)
print(5 not in list)
print(12 in list)
```

```
False
True
True
```

Example: Lottery

Let's assume that you've chosen the following numbers in the lottery: 3, 7, 11, 42, 34, 49. The numbers that have been drawn are: 5, 11, 9, 42, 3, 49. How many numbers have you hit?

```
drawn = [5,11,9,42,3,49]
bets = [3,7,11,42,34,49]
hits = 0
for number in bets:
    if number in drawn:
        hits +=1
print(hits)
```

```
4
```

L7.2 Tuples

L7.2.1 What is a tuple?

Like a list, a *tuple* allows to store several values in a single variable. It usually stores different types (heterogeneous) of data. It can behave like a list, but **it must not be modified at any time**: tuples are *immutable*. So, the programmer need not worry about any alterations to the tuple elements during the execution.

The first and the clearest distinction between lists and tuples is the syntax used to create them: tuples use parenthesis, whereas lists use brackets, although it's also possible to create a tuple just from a set of values separated by commas.

```
list1 = [1, 3, 5, 4, 2]     # This is a list
tuple1 = (1, 2, 4, 8)       # This is a tuple
tuple2 = 1., .5, .25, .125 # This is also a tuple
print(list1)
print(tuple1)
print(tuple2)
```

```
[1, 3, 5, 4, 2]
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
```

L7.2.2 Creating empty/one item tuple

Creating an empty tuple requires the parenthesis. When creating a one-element tuple, it needs to be differentiated from a variable. This is done by using parenthesis and/or a comma:

```
emptytuple = ()
oneelement1 = (1,)
oneelement2 = 1.,
print(emptytuple)
print(oneelement1)
print(oneelement2)
```

```
()
(1,)
(1.0,)
```

L7.2.3 Reading elements of a tuple

To get the elements of a tuple in order to read them over, use the same conventions as when using lists:

`tuplename[index]`

```
tuple = (1,10,100,1000)
print(tuple[0])
print(tuple[-1])
print(tuple[1:])
print(tuple[:-2])
for element in tuple:
    print(element)
```

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
```

L7.2.4 Don't try to modify tuple elements!

Tuples are immutable. So trying to modify them causes a runtime error. The following snippets show typical mistakes for tuple operations.

```
tuple = (1, 10,100,100)
tuple.append(10000) # Causes a runtime error
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-29-7c2b230be7d7> in <module>
  1 tuple = (1, 10,100,100)
->2 tuple.append(10000) # will cause a runtime error


AttributeError: 'tuple' object has no attribute 'append'
```

```
del tuple[0] # Causes a runtime error
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-33-c6c926d64047> in <module>
->1 del tuple[0] # Causes a runtime error


TypeError: 'tuple' object doesn't support item deletion
```

```
tuple[1] = -10 # Causes a runtime error
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-34-c33fcab7587e> in <module>
->1 tuple[1] = -10 # Causes a runtime error


TypeError: 'tuple' object does not support item assignment
```

L7.2.5 Joining tuples

The + operator, can concatenate (join) two tuples, yielding a new tuple containing all the elements from its arguments:

```
tuple1 = (1,10,100,1000)
tuple2 = 0,
print(tuple1 + tuple2)
```

```
(1, 10, 100, 1000, 0)
```

L7.2.6 What else can tuples do?

- The `len()` function accepts tuples, and returns the number of elements it contains.
- The `+` operator can join tuples together.
- The `*` operator can multiply a tuple by an integer `n` by joining `n` copies of the tuple.
- The `in` and `not in` operators work in the same way as in lists.
- A `for` loop can traverse all the elements of a tuple: it is a *sequence* type.

```
tuple = (1, 10,100)
t1 = tuple + (1000, 10000)
t2 = tuple * 3
print(t1)
print(t2)
print(10 in tuple)
print(-10 not in tuple)
for v in t2:
    print(v,end="-")
```

```
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
1-10-100-1-10-100-1-10-100-
```

One of the most useful tuple properties is their ability to appear on the left side of the assignment operator. It is commonly used for swapping values stored by variables. The following snippet shows two tuples interacting; the values stored in the variables "circulate": `v1` becomes `v2`, `v2` becomes `v3`, and `v3`becomes `v1`. The example presents one more important fact: a tuple's elements can be variables, not only literals. Moreover, they can be expressions if they're on the right side of the assignment operator.

```
v1 = 1
v2 = 2
v3 = 3
v1, v2, v3 = v2, v3, v1
print(v1, v2, v3)
```

```
2 3 1
```

L7.3 Dictionaries

L7.3.1 What is a dictionary?

A dictionary contains entries in the form of `key:value` pairs.

```
dct = {'cat': 'gato', 'dog':'perro', 'horse': 'caballo'}
phones = {'boss':5551233333, 'Ann':5552332123}
empty = {}
print(dct)
```

```
print(phones)
print(empty)
```

```
{'cat': 'gato', 'horse': 'caballo', 'dog': 'perro'}
{'Ann': 5552332123, 'boss': 5551233333}
{}
```

In the above snippet, for example, `'cat'` represents a key while `'gato'` is the value. In the first example, the dictionary uses keys and values which are both strings. In the second one, the keys are strings, but the values are integers. The reverse layout (keys → numbers, values → strings) is also possible, as well as number →number combination. The list of pairs is surrounded by curly braces, while the pairs themselves are separated by commas, and the keys and values by colons.

L7.3.2 Accessing elements in a dictionary

The values are accessed using a **valid** key.

```
print(dct['cat'])
print(phones['Ann'])
```

```
gato
5552332123
```

Getting a dictionary's value resembles indexing. If the key is a string, it must be specified as a string. Also, keys are case-sensitive: 'Ann' is something different from 'ann'.

Using a non-existent key causes a runtime error:

```
print(phones['president'])
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-56-4c8bd66313a2> in <module>
->1 print(phones['president'])


KeyError: 'president'
```

Testing existence of a key can be done using the `in` keyword:

```
print('president' in phones)
print('Ann' in phones)
print(5552332123 in phones)  # The value 5552332123 exists but it looks␣
↪for keys
```

```
False
True
False
```

L7.3.3 Traversing a dictionary with a for loop

A dictionary cannot be traversed directly with a `for` loop: is not a sequence type like list and tuple. But there are simple and very effective tools that can adapt any dictionary to the `for` loop requirements.

The first of them is a method named `keys()`, owned by each dictionary. The method returns a list containing all the keys gathered within the dictionary. Having a list of keys enables to access the whole dictionary in an easy and handy way.

```
dct = {'cat': 'gato', 'dog':'perro', 'horse': 'caballo'}
for key in dct.keys():
    print(key, '→', dct[key])
```

```
cat → gato
horse → caballo
dog → perro
```

Another way is based on using a dictionary's method named `items()`. The method returns a list of tuples where each tuple is a key–value pair.

```
dct = {'cat': 'gato', 'dog':'perro', 'horse': 'caballo'}
for en,sp in dct.items(): # Each turn, the tuple (en,sp) is used as the␣
↪left handside of the assignment
    print(en,'→',sp)
```

```
cat → gato
horse → caballo
dog → perro
```

There is also a method named `values()`, which works similarly to `keys()`, but returns a list of values.

```
dct = {'cat': 'gato', 'dog':'perro', 'horse': 'caballo'}
for sp in dct.values():
    print(sp)
```

```
gato
caballo
perro
```

L7.3.4 Assigning a new value to an existing key

Assigning a new value to an existing key is simple. There are no obstacles to modifying dictionaries: dictionaries are fully *mutable*. For example, to replace the value "gato" with "minou":

```
dct = {'cat': 'gato', 'dog':'perro', 'horse': 'caballo'}
dct['cat'] = 'minou'
print(dct)
```

```
{'cat': 'minou', 'horse': 'caballo', 'dog': 'perro'}
```

L7.3.5 Adding a new key-value pair

Assign a value to a new, previously non-existent key adds a new key-value pair to the dictionary. This is very different behavior compared to lists, which don't allow assignment of values to non-existing indices. For example, to add a new pair of words: `lion→leon` to the dictionary:

```
dct['lion'] = 'leon'
print(dct)
```

```
{'lion': 'leon', 'cat': 'minou', 'horse': 'caballo', 'dog': 'perro'}
```

L7.3.6 Removing a key-value pair

Removing a key causes the removal of the associated value. Values cannot exist without their keys. This is done with the `del` instruction.

```
del dct['dog']
print(dct)
```

```
{'lion': 'leon', 'cat': 'minou', 'horse': 'caballo'}
```

Trying to remove a non-existent key causes a runtime error:

```
del dct['bird']
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-54-963ba9b2e447> in <module>
->1 del dct['bird']


KeyError: 'bird'
```

L7.3.7 Example: Searching for certain words

In the following code, `dct` is a dictionary of English words to Spanish translations. The program ask the user to search for some words and provides the translation if the word exists in the dictionary.

```
dct = {'cat': 'gato', 'dog':'perro', 'horse': 'caballo'}

word = input('Enter an English word (empty string to stop): ')
while word != '':
    if  word in dct:
        print(word, '→',dct[word])
    else:
        print('Not in dictionary:', word)
```

```
    word = input('Enter an English word (empty string to stop): ')
```

```
Enter an English word (empty string to stop): cat
cat → gato
Enter an English word (empty string to stop): dog
dog → perro
Enter an English word (empty string to stop): house
Not in dictionary: house
Enter an English word (empty string to stop):
```

L7.4 Sequence data type and mutability

Lists, tuples, and dictionaries are data types in which you can store more than one value. They differ fundamentally in regard of their *sequence type* character and *mutability*. While tuples and lists are sequence data structures, dictionaries are not. While lists and dictionaries are mutable, tuples are not.

Let's recapitulate what these two concepts are.

A *sequence* type is a type of data in Python which is able to store more than one value. These values can be sequentially browsed; in other words, a sequence is a data type which can be scanned by a `for` loop. List is an example of sequence data.

Mutability is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: mutable and immutable. Mutable data can be freely updated at any time. For example, the following operation can update a list at any time (it appends 1 to the end of the list):

```
list.append(1)
```

Immutable data cannot be modified in this way. For example, tuple data is immutable.

Lecture 8
Functions

Lecture contents

L8.1 Introduction **67**
L8.2 Function definition **67**
L8.3 Calling a function **68**
L8.4 Examples **69**
L8.4.1 Modularity: cleaning the code using functions 69
L8.4.2 Maximum function 69
L8.4.3 A function with no parameters 70
L8.4.4 A function with no `return` statement 70
L8.4.5 A `main` function 71
L8.5 Variable Scope in Functions **71**
L8.5.1 An illustrated example of variable scoping 73
L8.5.2 Summary of scope concepts 74
L8.6 Recursion **74**
L8.6.1 An illustrated example of recursion 75
L8.6.2 Example: Factorial 77
L8.6.3 Example: Triangular numbers 79
L8.7 Named parameters **79**
L8.8 Functions in math vs Python **80**
L8.8.1 (Non)determinism 80
L8.8.2 Purity and side-effects 81

L8.1 Introduction

In order to do the same or a similar task repeatedly one approach would be to copy and paste blocks code over and over again. But it makes the program more cluttered and the code difficult to maintain: correcting a bug must be done in all the copies! A better approach is to split the code into multiple *functions* dedicated each to a task. That is the informal definition of a function in Python: a block of code which executes a specific task. Code written using this method is more compact and manageable.

L8.2 Function definition

Function definition is performed using the following syntax:

```
def function_name(parameters):

  # Function code (body)

  return return_value
```

It has the following elements:

- All function definition start with the keyword `def`.
- Then comes the function name: an identifier which will be used to execute the function code. As for variable names, it is better to choose significant function names that describe what the code does.
- The `parameters` are a comma-separated list of variables, used as an input to the function. Some functions don't have any parameter.
- The first line, which must end with a colon (`:`), gives the *signature* of the function.
- Below, as an **indented block** is the body of the function: the block of code to be executed. Inside this block, the parameters are used as variables, assuming they have a value.
- In the body there can be `return` statements that terminate the function and return a value. Some function don't return anything.

For example, a function which returns the area of a rectangle requires two variables `width` and `length` as *parameters* (inputs). A good function name can be `area`, since it is what the function calculates. So the code starts with `def area(width,length):`. Then the body of the code is the actual calculation of the surface area in a variable that can be returned:

```
  a = width*length
  return a
```

Putting everything together:

```
def area(width, length):
  a = width * length
  return a
```

L8.3 Calling a function

The function `area` does not do anything until it is *called*. This is why the code above does not produce any result.

To call a function, use its name and provide a value for each parameter in the order they were declared in the function definition. For example:

```
area(2,3)
```

```
6
```

The above line calls the `area` function with `width = 2` and `length = 3`. This expression is replaced by the value returned by the function, in this case 6.

The result of functions can be stored in variables or used in calculations:

```
result = area(2,3)
doubleA = 2*area(2,3)
```

The line `result = area(2,3)` assigns the returned value to a variable `result`. It can then be printed with `print(result)`.

```
result = area(2, 3) # Function call, store the returned value in a␣
↪variable
print(result)       # Print the result
```

```
6
```

L8.4 Examples

L8.4.1 Modularity: cleaning the code using functions

In the following code, computing the average of two numbers is done twice.

```
x = 3
y = 4
avg = (x+y)/2.0
print(avg)
x = 5
y = 7
avg = (x+y)/2.0
print(avg)
```

```
3.5
6.0
```

By defining an `avg` function, the code can be simplified. Since the average needs two values to average them, the `avg` function requires two parameters: `x` and `y`.

```
def avg(x,y):
  return (x+y)/2.0

print(avg(3,4)) # Function call
print(avg(5,7)) # Functions are defined once but can be called several␣
↪times. This is the second function call.
```

```
3.5
6.0
```

L8.4.2 Maximum function

The body of the function can take many forms, using all the code structures available in Python. In this example, the `maximum` function, which returns the maximum of given two numbers, `x` and `y`, uses a conditional.

See Lecture L5 for more on conditionals.

It can be designed using the following procedure (in English, not in Python):

```
Define a maximum function with two parameters x and y
  If x is greater than y, return x
  Otherwise, return y
```

Which can be translated into Python as follows:

```python
def maximum(x,y): # Define the function's signature
    if x>y:       # "If x is greater than y"
      return x
    else:         # "Otherwise"
      return y

print(max(3,2))   # Function call
print(max(2,3))   # Function call
```

```
3
3
```

L8.4.3 A function with no parameters

It is perfectly possible that a function requires no parameters. For example, the function below manages input from the user and returns it, but takes no parameter:

```python
def getName():   # No parameters, the parentheses are still here
    s = ""
    while s=="": # Continue until user input is not empty
        s = input("Please enter your name: ")
    return s

n = getName()    # Call the function, store result in variable n
print("Hello,",n)
```

```
Please enter your name:
Please enter your name:
Please enter your name: John
Hello, John
```

L8.4.4 A function with no `return` statement

Similarly, some functions do not return any value. In this case the function stops when all the code has been executed. It is also possible to have the **`return`** keyword with no value afterwards.

```python
def sillyFunction(n,k):
    if n=="":
        print("You have no name")
        return # Stops the function here
    for i in range(k,0,-1):
        print("Let me tell you",i,"more times: Hello",n,"!")
```

```
    #return # This could be uncommented without any effect

sillyFunction("Bob",5)
sillyFunction("",42)
```

```
Let me tell you 5 more times: Hello Bob !
Let me tell you 4 more times: Hello Bob !
Let me tell you 3 more times: Hello Bob !
Let me tell you 2 more times: Hello Bob !
Let me tell you 1 more times: Hello Bob !
You have no name
```

L8.4.5 A `main` function

Functions that neither take parameters nor return anything are quite rare, to the exception of the `main` function, which is the name usually given to the main function of the program: its role is to call other functions.

```
def getNumber():
    i = -1
    while i<0:
        i = int(input("Please enter a non-negative number: "))
    return i

def main():
    name = getName()             # Get user input
    number = getNumber()         # Get user input
    sillyFunction(name,number) # Do some printing

main() # Launch the whole program
```

```
Please enter your name:
Please enter your name: Alan
Please enter a non-negative number: -12
Please enter a non-negative number: 3
Let me tell you 3 more times: Hello Alan !
Let me tell you 2 more times: Hello Alan !
Let me tell you 1 more times: Hello Alan !
```

L8.5 Variable Scope in Functions

Parameters and variables defined inside functions only exist within the function. This allows for two different functions to use the same variable name without any issue, but generates an error if a variable is used outside of its *scope*.

```
def myFunction():
    localVariable = 100
    return
```

```
print(localVariable) # Does not exist here: "NameError: name␣
↪'localVariable' is not defined"
```

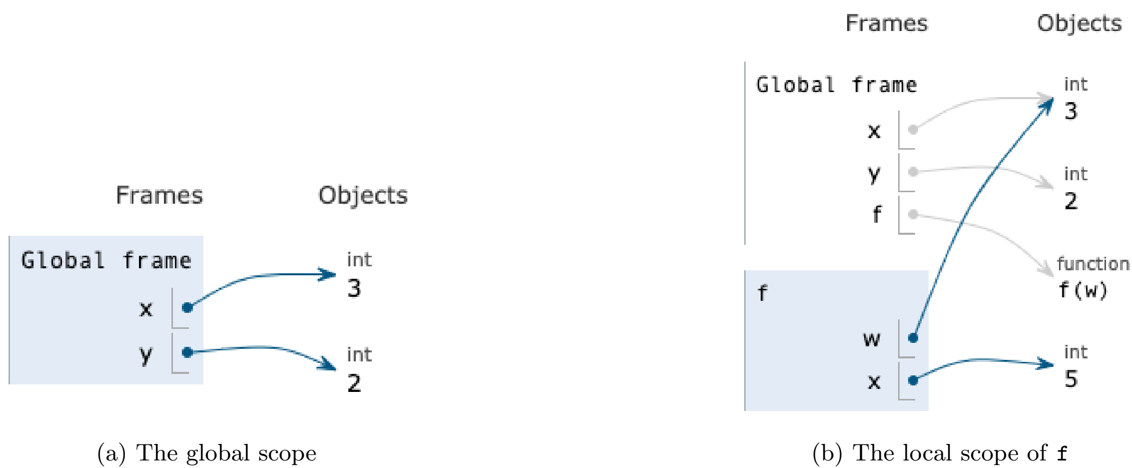
```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-1-bd6ab172a552> in <module>
    2     localVariable = 100
    3     return
->4 print(localVariable) # Does not exist here: "NameError: name␣
↪'localVariable' is not defined"


NameError: name 'localVariable' is not defined
```

Global variables are variables defined outside of functions. They can be accessed provided the function declares that it uses the global version using the `global` keyword.

```
x = 42       # Global variable x
def aFunction():
    x = 100 # Local variable x; not the same x despite the same name!
    print(x)
    return

print(x)
aFunction()
print(x)      # The local value does not exist anymore, using the global␣
↪variable
```

```
42
100
42
```

```
x = 42        # Global variable x
def anotherFunction():
    global x # Declaring that the x used in this function will be the␣
↪global one
    x = 100  # Assigns the global variable
    print(x)
    return

print(x)
anotherFunction()
print(x)      # Using the global variable (which value was changed by the␣
↪function call)
```



(a) The global scope

(b) The local scope of `f`

Figure L8.1: Illustration of variable scoping.

```
42
100
100
```

L8.5.1 An illustrated example of variable scoping

Consider the code below. The printed result is `3 5`; let's see why. Note: the illustrations of Figure L8.1 can be viewed dynamically on the Python tutor website.

```
x = 3
y = 2

def f(w):
  x = w+y
  return x

y = f(x)
#print(w) # Would produce an error since local w is called outside f.
print(x,y)
```

```
3 5
```

```
x=3
y=2
```

First, **global** variables `x` andy are defined, so `x` and `y` can be accessed everywhere including inside of `f` (see Figure L8.1(a)).

```
y = f(x)
```

The `y = f(x)` calls the function `f` with `w = 3` since `x = 3` and assigns the returned value to `y`. Let us look at the inside of `f` carefully.

```
x = w+y
```

This calculation uses the **local** variable `w = 3` and the **global** variable `y = 2`; their sum `w+y` is `5`. A new **local** variable `x` is defined inside `f` since `f` cannot modify the **global** variable `x`; it is assigned the value `5`.

Eventually, `f` returns the value of `x`(local) which is `5`.

So `5` is assigned to `y` (global). The code is outside `f`, so **local** variables `x` and `w` (that were defined inside `f`) cannot be accessed anymore. So `print(w)` outside `f` would result in an error.

Here, the main point is that the global variable `x` is still `3`. Therefore, `print(x,y)` prints

```
3 5
```

L8.5.2 Summary of scope concepts

Let us summarize the above findings.

- **local** variable: a variable defined **inside** a function.
- **global** variable: a variable defined **outside** a function.
- Inside a function, a **global** variable can be accessed, but it cannot be modified. So, when it is assigned, a computer assigns it to a newly defined **local variable** instead.
- Outside a function, a **local** variable cannot be accessed.

Note: There are other scopes beside **local** and **global**, but they are "beyond the scope" of this course.

L8.6 Recursion

As in previous examples, it is possible to call a function from within a function. Let's visualize what happens in that case.

```
def f(x):
  return 3*x

def g(x):
  y = f(x)
  return y+1

print(g(1))
```

```
4
```

Function `g` calls function `f`. So it waits for function `f` to terminate and replaces the value `f(x)` by the value returned by `f`, as depicted in Figure L8.2.

It is also possible for a function to call not another function, but itself. This is called, *recursion*.

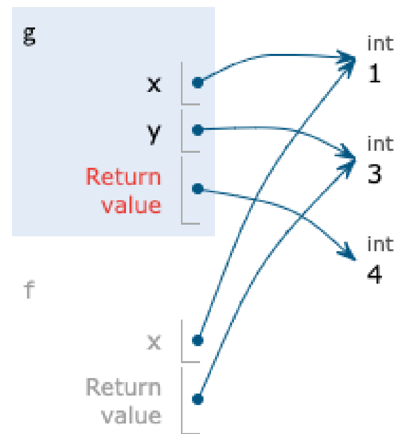


Figure L8.2: Illustration of function `g` calling function `f`.

L8.6.1 An illustrated example of recursion

```
def f(x):
  if x==1:
    y = 1
  else:
    y = f(x-1)+1
  return y

z = f(3)
print(z)
```

```
3
```

Let's visualize what happens when `f(3)` is called (Figure L8.3(a)).

Case `x = 3`

Since $x = 3 \neq 1$, the `else` is taken and the calculation is as follows: `f(3) = f(3-1)+1 = f(2)+1`

So in order to compute `f(3)`, the function calls `f(2)` (Figure L8.3(b)).

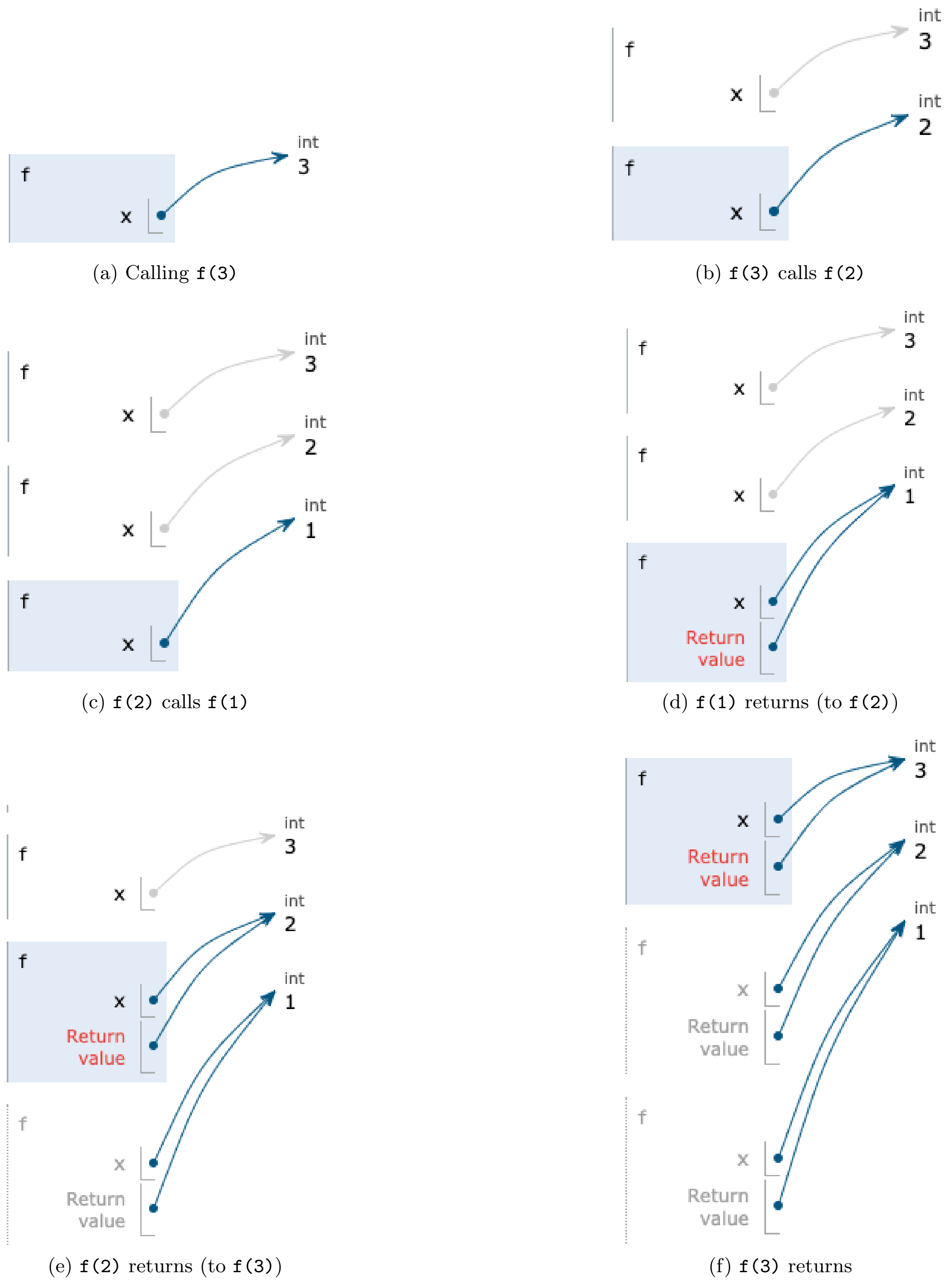
Case `x = 2`

Since $x = 2 \neq 1$, the `else` is taken and the calculation is as follows: `f(2) = f(2-1)+1 = f(1)+1`

So in order to compute `f(2)`, the function calls `f(1)` (Figure L8.3(c)).

Case `x = 1`

Since $x = 1$, the *then* part of the condition is executed and the function returns `1` (Figure L8.3(d)).



(a) Calling `f(3)`

(b) `f(3)` calls `f(2)`

(c) `f(2)` calls `f(1)`

(d) `f(1)` returns (to `f(2)`)

(e) `f(2)` returns (to `f(3)`)

(f) `f(3)` returns

Figure L8.3: Illustration of a *recursive* call.

Going back to the original call

After `f(1)` returns, `f(2)` can finish its calculation and return `2` (Figure L8.3(e)). Then `f(3)` can finish it's calculation and return `3` (Figure L8.3(f)).

Remarks

- Since the `return` stops the execution of the function, it is possible to omit the `else` part, although it may become less readable:

```
def f(x):
    if x==1:
        return 1      # If this line is executed
    return f(x-1)+1 # this one will not be
```

- Each time `f` is called from within `f`, it is another copy of the code of `f` that is executed. Because of that, here is a limitation on recursion depth that is supported by Python. It is usually set at 10000 by default.
- This whole sequence can be visualized dynamically on the Python tutor website.

L8.6.2 Example: Factorial

The factorial function is defined by

$$factorial(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Since for $n > 1$

$$factorial(n-1) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1),$$

the definition can be rewritten as

$$factorial(n) = n \cdot factorial(n-1)$$

when $n > 1$.

This provides a recursive relation for the factorial:

$$\begin{cases} factorial(n) = 1, & n \leq 1 \\ factorial(n) = n \cdot factorial(n-1), & n > 1 \end{cases}$$

is equivalent to the original definition of factorial.

The above relation translates quite directly into Python code:

```python
def factorial(n):
  if n<=1:
    return 1
  else:
    return n*factorial(n-1)

print(factorial(3))
print(factorial(5))
```

```
6
120
```

The factorial function could also have been coded in Python using loops, in a style which is called *iterative*.

To get to the iterative version from the recursive version, consider that factorial is called first for `n`, then for `n-1` all the way down to `1`, but the calculations themselves are done in the other direction: `factorial(1)` returns then `factorial(2)` returns, all the way to `factorial(n)`. Each intermediate result could be stored in a variable, used by the next step.

For example for `factorial(3)`:

```
f1 = 1
i = 2
f2 = f1*2
i = 3
f3 = f2*3
```

which means

```
f1 = 1
i = 2
f2 = f1*i
i = 3
f3 = f2*i
```

Variables `f1, f2, f3` can all be replaced by a single variable `f`. At the end of the program, `f` s the value of `f3`.

```
f = 1
i = 2
f = f*i
i = 3
f = f*i
```

Since the instruction `f=f*i` is repeated multiple times by increasing `i` by `1`, this sequence can be replaced by a loop:

```
f = 1
for i in range(2,4): # Up to 3
  f = f*i
```

This can be generalized to write an iterative version of `factorial(n)`. Note that it resembles the first definition that was given of the factorial function.

```
def factorial(n):
  f = 1
  for i in range(2,n+1): # Up to n (included)
    f = f*i
```

```
  return f

print(factorial(3))
print(factorial(5))
```

```
6
120
```

When programming recursively, no loops are involved, but there is hidden repetition trough the calling of copies of the function. This repetition can take up a lot of memory. As a result, although recursive functions are quite elegant, they may not be very efficient.

L8.6.3 Example: Triangular numbers

The *triangle*(n) function is defined by

$$triangle(n) = 1 + 2 + 3 + 4 + \ldots + n$$

It can also be defined recursively:

$$\begin{cases} triangle(n) = 1, & n = 1 \\ triangle(n) = n + triangle(n-1), & n > 1 \end{cases}$$

The corresponding `triangle(n)` function can be defined in Python either iteratively or recursively. (The two functions were given different names in order to distinguish the programming style used.)

```
def triangleIter(n):
    t = 0
    for i in range(1,n+1):
        t = t+i
    return t

def triangleRec(n):
    if n==1:
        return 1
    else:
        return n+triangleRec(n-1)

print(triangleIter(4))
print(triangleRec(4))
```

```
10
10
```

L8.7 Named parameters

When calling a function, all parameters have to be given a value, and passed in the order parameters were declared. This is not the case for *named parameters*. The named parameters

are defined after the *positional parameters* (the regular ones, denoted as positional because what value corresponds to what parameter depends on the position), and are given not only a variable name but also a default value.

When calling a function with named parameters, they also come at the end but the order within named parameters can be changed. In addition, if arguments for named parameters are not given, then the default value is used.

```
def theaterLine(speaker,text,indentation=1,didascalia=""):
    indent = "\t"*indentation
    text = text.replace("\n","\n"+indent)
    info = ""
    if didascalia!="":
        didascalia = "\t\t/"+didascalia+"/"
    print(speaker+":"+didascalia+"\n"+indent+text)
    return

# No argument passed to named parameters, using default values
theaterLine("Richard, Duke of Gloster","Now is the winter of our␣
↪discontent\nMade glorious summer by this son of York")
print() # New line
# Named parameters given in any order
theaterLine("Algernon","I don't know that I am much interested in your␣
↪family life, Lane.",didascalia="Languidly",indentation=2)
```

```
Richard, Duke of Gloster:
        Now is the winter of our discontent
        Made glorious summer by this son of York

Algernon:               /Languidly/
                I don't know that I am much interested in your family␣
↪life, Lane.
```

L8.8 Functions in math vs Python

L8.8.1 (Non)determinism

In mathematics, the definition of the term *function* states that

$$f \text{ is a function if for each } x,\ f \text{ returns a } \textbf{unique } f(x).$$

For example, if $f(x) = 2 \cdot x + 1$, the corresponding Python `f` function is given below:

```
def f(x):
  return 2*x+1

print(f(1))
print(f(1))
```

```
3
3
```

Every time $f(1)$ is calculated, $f(1) = 3$. This is called *determinism*.

This restriction is not true for a Python function: it does not need to return the same value for a given `x`, because it may rely on data outside the scope of the function.

```
from random import randint
def g(x):
  y = randint(1,10) # y is a random value
  return x+y

print(g(1))
print(g(1))
```

```
3
6
```

Here, `randint(1,10)` returns an integer between 1 and 10, chosen randomly. So `print(g(1))` may return a different number each time.

Example: Nondeterminism through user input

Another source on nondeterminism can be the user input.

```
def nonDetInput(x):
  y = int(input("Type an integer y: "))
  return x+y

print(nonDetInput(7))
print(nonDetInput(7))
```

```
Type an integer y: 42
49
Type an integer y: -37
-30
```

The above function is not deterministic since it returns a different value depending on the user input `y`. For example, if the user types `42`, `f(7)` returns `49`. But if the user types `-37`, `f(7)` returns `-30`.

L8.8.2 Purity and side-effects

Another restriction on a mathematical function is that it *only calculates*. It cannot modify anything that is outside its internal workings: calculating $f(x)$ cannot change y or x itself.

This restriction again does not apply to functions in programming languages: these can modify objects outside of their scope. This is called a *side-effect*.

One relatively benign but widespread side-effect is the printing of some output: it changes the state of the terminal (or other output stream), and as such is considered a side-effect.

More involved side effects include the modification of a global variable, as is done in Section L8.5. Since global variables are rarely used, this case is not so widespread.

More on mutability in Section L7.4.

More common is the modification of *mutable* objects passed as arguments (such as lists and dictionaries), as in the example below. The `rotate` function takes the first element of the list and places it at the very end.

```
def rotate(l):
    if len(l)>1:
        l.append(l[0])
        del l[0]

somelist = [1, 2, 3, 4]
print(somelist)
rotate(somelist)
print(somelist)
rotate(somelist)
print(somelist)
rotate(somelist)
print(somelist)
```

```
[1, 2, 3, 4]
[2, 3, 4, 1]
[3, 4, 1, 2]
[4, 1, 2, 3]
```

In computer science, a function with no side-effect, for example any mathematical function, is called a *pure function*. Pure functions are easier to debug because there is only one way they can influence the rest of the program, which is through the `return` instruction. Side-effects of non-pure functions are harder to debug because executing the function several times may not be equivalent to running it only once, and it may look like it is non-deterministic!

Part II

Projects

Project 1

The Motion of the Vertical Projectile

Project contents

P1.1 Top height 85
P1.2 Flight time 86
P1.3 Possible extensions 87

Consider the following experiment: Throw a rock vertically. There is an initial height of the ground (`h0` "h zero") and initial velocity `v0`. The rock climbs up to certain height. As it does so, its velocity decreases due to the force of attraction (gravity) exerted by Earth. When the rock reaches its top height, its velocity is zero. Afterwards, the rock falls down and eventually hits the ground.

The formula for the velocity of the object t seconds after being thrown is given by:

$$v(t) = gt + v_0 \ . \tag{P1.1}$$

The formula for the height of the object t seconds after being thrown is given by:

$$h(t) = \frac{g}{2}t^2 + v_0 t + h_0 \ . \tag{P1.2}$$

The time t_{top} after which the object reaches its top height satisfies $v(t_{top}) = 0$. It can be calculated as:

$$t_{top} = -\frac{v_0}{g} \ . \tag{P1.3}$$

The top height attained by the object is given by:

$$h_{top} = h(t_{top}) = h_0 - \frac{v_0^2}{2g} \ . \tag{P1.4}$$

P1.1 Top height

This first code computes the top height h_{top} reached by a vertically thrown rock:

```python
g = -9.81 # the gravity or acceleration -9.81 meters/second squared

def top_height(h0, v0):
    return h0-(v0**2)/(2*g)

def main():
    print('Computing the height in meters of an object',
          'with initial height h0 and velocity v0:')
    while True:
        s = input('Press q to quit or anything else to continue: ')
        if s == 'q':
            break
        h0 = input('Enter initial height: ')
        h0 = float(h0)
        v0 = input('Enter initial velocity: ')
        v0 = float(v0)
        toph = top_height(h0,v0)
        print('top height = %.2f meters.' % toph)
main()
```

```
Computing the height in meters of an object with initial height h0 and␣
 ↪velocity
v0:
Press q to quit or anything else to continue: w
Enter initial height: 10
Enter initial velocity: 50
top height = 137.42 meters.
Press q to quit or anything else to continue: w
Enter initial height: 1.8
Enter initial velocity: 340
top height = 5893.75 meters.
Press q to quit or anything else to continue: q
```

P1.2 Flight time

The time t_{air} after which the object reaches the ground satisfies $h(t_{air}) = 0$. It can be calculated to be:

$$t_{air} = -\frac{v_0 + \sqrt{v_0^2 - 2gh_0}}{g} \ . \tag{P1.5}$$

This code computes the time the vertically thrown rock is in air (or the flight time):

```python
g = -9.81 # The gravity or acceleration: -9.81 meters/second squared
import math
def time_in_air( h0, v0):
    return -(v0+math.sqrt(v0**2-2*g*h0))/g
```

```python
def main():
    print('Computing the height in meters of an object',
          'with initial height h0 and velocity v0:')
    while True:
        s = input('Press q to quit or anything else to continue: ')
        if s == 'q':
            break
        h0 = input('Enter initial height: ')
        h0 = float(h0)
        v0 = input('Enter initial velocity: ')
        v0 = float(v0)
        tair = time_in_air(h0,v0)
        print('Time in air is = %.2f seconds.' % tair)
main()
```

```
Computing the height in meters of an object with initial height h0 and␣
 ↪velocity
v0:
Press q to quit or anything else to continue: w
Enter initial height: 1.8
Enter initial velocity: 340
Time in air is = 69.32 seconds.
Press q to quit or anything else to continue: q
```

P1.3 Possible extensions

The project may be expanded so that all the formulas are implemented in their respective functions.

Project 2
Linear and Quadratic Equations

Project contents

P2.1 Solving linear equations 89
P2.1.1 Example: Solve $2x - 4 = 0$ for x 89
P2.1.2 Generalization with parameters a and b 89
P2.1.3 Case $a = 0$ 90
P2.1.4 Coding the procedure in Python 90
P2.2 Quadratic equations 92
P2.2.1 Hints to solve the problem 92
P2.2.2 Project Code 95

P2.1 Solving linear equations

A *linear equation* is an equation of the form $ax + b = 0$ to be solved for x, where a and b are real numbers.

P2.1.1 Example: Solve $2x - 4 = 0$ for x

$$\begin{aligned} 2x - 4 &= 0 \\ 2x &= 4 \\ x &= \tfrac{4}{2} \\ x &= 2 \end{aligned}$$

Here 4 was added to both sides, then they were divided by 2 to find x.

P2.1.2 Generalization with parameters a and b

The above computation can be generalized with parameters a and b.

$$\begin{aligned} ax + b &= 0 \\ ax &= -b \\ x &= -\tfrac{b}{a} \end{aligned}$$

Here b was added to both sides which were then divided by a to find x. The problem here is that a may be zero, and **division by zero is undefined**! Therefore the division by a can only be performed when $a \neq 0$: this needs to be tested in a conditional statement.

An English version this procedure can be devised: (It is **not** Python code!)

```
if a is not zero:
    x is -b/a
    print x
else: (that means a is zero)
    handle it carefully
```

P2.1.3 Case $a = 0$

Now let's consider the case when $a = 0$. The equation is now $0x + b = 0$, which is equivalent to $b = 0$. So x does not appear in this equation, and it's value is not relevant, only the value of b is.

- If $b = 0$, then $ax + b = 0$ becomes $0 = 0$ which is always *True* regardless of x: all real numbers are solutions.
- If $b \neq 0$, then $ax + b = 0$ becomes $b = 0$ which is impossible (it is in contradiction with $b \neq 0$). So there is no x satisfying $ax + b = 0$: there is no solution.

Therefore **another condition** is needed to handle these two cases.
The procedure can therefore be refined as follows:

```
if a is not zero:
    x is -b/a
else:  (that means a is zero)
    if b is zero:
        x is "All solutions"
    else:
        x is "No solution"
print x
```

A graphical view of the 3 possible cases in linear equations is depicted in Figure P2.1.

P2.1.4 Coding the procedure in Python

The goal now is to convert the procedure written in English, also known as an *algorithm*, into Python code that can be executed.
The end result should look like these examples:

- Case with one solution:

```
Type a real number a: 2
Type a real number b: 3
x = -1.5
```

- Case with no solution:

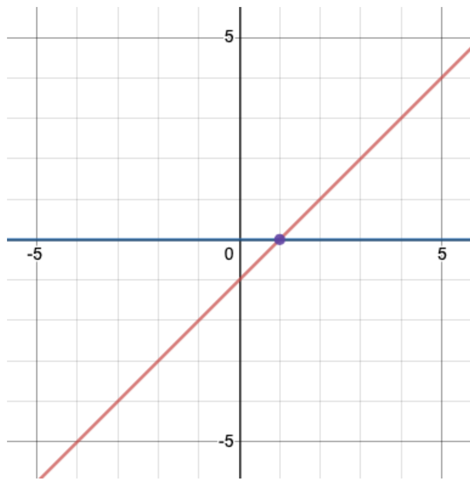
```
Type a real number a: 0
Type a real number b: 2
No solution
```

- Case with all reals as solutions:

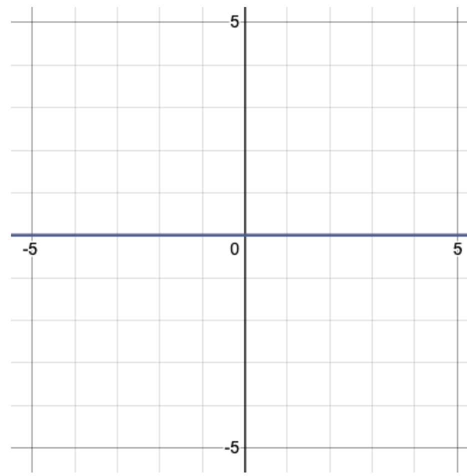
```
Type a real number a: 0
Type a real number b: 0
All solutions
```

Note

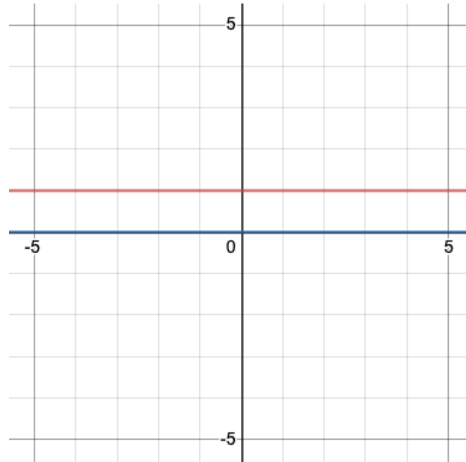
Since `a` and `b` are **real numbers**, so the user input for these values need to be converted into a `float`.



(a) One solution ($a \neq 0$)



(b) All solutions ($a = 0,\ b = 0$)



(c) No solution ($a = 0,\ b \neq 0$)

Figure P2.1: A graphical view of the 3 possible cases in linear equations

```
a = float(input("Type a real number a:"))
```

Now complete the code below:

```
# Linear equation (ax+b=0) solver
a = float(input("Type a real number a: ")) # float: real numbers
b = float(input("Type a real number b: ")) # float: real numbers
#
# Complete the code here
#
```

P2.2 Quadratic equations

A *quadratic equation* is an equation of the form $ax^2 + bx + c = 0$ to be solved for x, where a, b and c are real numbers. It is also assumed that x is a real number.

Solving such an equation can be done using the quadratic formula.

If $a \neq 0$, the quadratic formula gives

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If $a = 0$, then the equation becomes $bx + c = 0$. So a similar procedure as in the case of Section P2.1 above can be used.

The goal is to code a Python program that takes values a, b, c, and prints out the solutions of the equation.

P2.2.1 Hints to solve the problem

Hint 1: $a = 0$ case

When $a = 0$, we have

$$bx + c = 0$$

It is a linear equation as in the Section P2.1. Although the idea remains the same, this time b is the coefficient of x, instead of a, and c now has the role that b had. The code must be modified in consequence.

Hint 2: Square root

To compute The quare root, use the `sqrt` function from `math` library. For example to compute $\sqrt{2}$:

```
from math import sqrt
sqrt(2)
```

```
1.4142135623730951
```

The `sqrt` function is only defined for non-negative numbers and produces an error on negative numbers:

```
sqrt(-4)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-4-ee0bfbfc0490> in <module>
->1 sqrt(-4)


ValueError: math domain error
```

So to have real solution(s), $b^2 - 4ac$ must be non-negative. In other words:

- If $b^2 - 4ac \geq 0$ there are real solutions.
- If$b^2 - 4ac < 0$, there is **no real solution**.

Hint 3: All possible cases from the quadratic formula with $a \neq 0$

When $a \neq 0$, there are actually three possible cases depending on the value of $b^2 - 4ac$.

If $b^2 - 4ac = 0$, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b \pm 0}{2a} = \frac{-b}{2a}$, so there is only one real solution:

$$x = \frac{-b}{2a}$$

If $b^2 - 4ac > 0$, there are two real solutions:

$$x_1 = \frac{\left(-b + \sqrt{b^2 - 4ac}\right)}{(2a)} \qquad x_2 = \frac{\left(-b - \sqrt{b^2 - 4ac}\right)}{(2a)}$$

As stated before, if $b^2 - 4ac < 0$, there is no real solution and the program should print `"No real solution"`.

A graphical view of the 3 possible cases in quadratic equations with $a \neq 0$ is depicted in Figure P2.2.

Hint 4: Be careful with division and use parentheses.

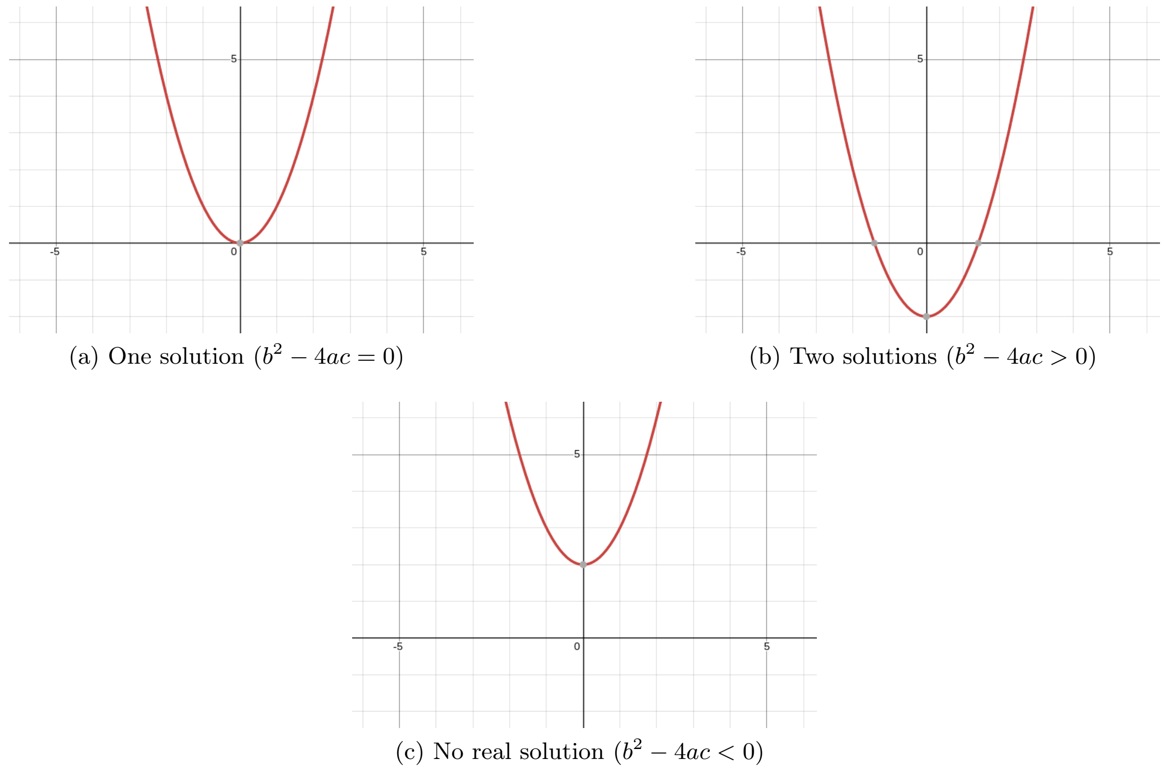
In mathematics, the fraction bar actually serves as parentheses. The value of

$$\frac{2x + 8}{3x}$$

with $x = 2$ is therefore

$$\frac{2 \times 2 + 8}{3 \times 2} = (2 \times 2 + 8) \div (3 \times 2) = 12 \div 6 = 2$$

In Python, actual **parentheses** have to be used: `(2*x+8)/(3*x)` produces the desired result. Below is an example of what happens when the parentheses are missing



(a) One solution ($b^2 - 4ac = 0$)

(b) Two solutions ($b^2 - 4ac > 0$)

(c) No real solution ($b^2 - 4ac < 0$)

Figure P2.2: A graphical view of the 3 possible cases in quadratic equations with $a \neq 0$

```
x=2
e1=2*x+8/3*x       # Interpreted as (2*x)+(8/3)*x
e2=(2*x+8)/3*x     # Interpreted as (((2*x)+8)/3)*x
e3=(2*x+8)/(3*x)   # Interpreted as ((2*x)+8)/(3*x)
print(e1,e2,e3)
```

```
9.333333333333332 8.0 2.0
```

Similarly,

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

is written `(-b+sqrt(b*b-4*a*c))/(2*a)` in Python.

Hint 5: Sample outputs

Below are examples of the expected output.

- Case with all reals as solutions:

```
Type a real number a: 0
Type a real number b: 0
Type a real number c: 0
All solutions
```

- Case of linear equation with no solution:

```
Type a real number a: 0
Type a real number b: 0
Type a real number c: 2
No solution
```

- Case of linear equation with a single solution:

```
Type a real number a: 0
Type a real number b: 2
Type a real number c: 3
x = -1.5
```

- Case of a quadratic equation with two solutions:

```
Type a real number a: 2
Type a real number b: -5
Type a real number c: -12
x1 = 4.0, x2 = -1.5
```

- Case of a quadratic equation with a single solution:

```
Type a real number a: 1
Type a real number b: 4
Type a real number c: 4
x = -2.0
```

- Case of a quadratic equation with no real solution:

```
Type a real number a: 1
Type a real number b: 2
Type a real number c: 5
No real solution
```

P2.2.2 Project Code

Complete the code of the project below:

```
# Quadractic equation (ax^2+bx+c=0) solver
from math import sqrt
a = float(input("Type a real number a: ")) # float: real numbers
b = float(input("Type a real number b: ")) # float: real numbers
c = float(input("Type a real number c: ")) # float: real numbers
#
# Complete the code here
#
```

Project 3
Image Processing

Project contents

P3.1 Introduction 97
P3.1.1 Colors and transparency 97
P3.2 Code preamble and first manipulation: input/output 98
P3.3 Image format 99
P3.3.1 Pixel organization 99
P3.3.2 Creating an Alpha channel 100
P3.3.3 Experiments with resolution and colors 100
P3.4 Changing images pixel by pixel 101
P3.4.1 Coloring the image by adding light 101
P3.4.2 Coloring the image by removing light 102
P3.4.3 Other color manipulations 103
P3.5 Global and vicinal manipulation 104
P3.5.1 Global brightness of an image 104
P3.5.2 Blurring an image 105
P3.6 Doing all transformations onto files 106
P3.6.1 Exercise: The wrap-up function 108

P3.1 Introduction

The objective of this project is to learn about the way images are coded, and apply image manipulation on them. The image format used here is Portable Network Graphics (PNG), which supports transparency. The manipulations include change in color and opacity.

P3.1.1 Colors and transparency

An image is made of (many) small squares called ***pixels*** (standing for *picture cell*), each having an individual color and transparency.

One way colored are coded in computer graphics is called the RGBA model, standing for Red, Green, Blue, Alpha. The Red, Green, and Blue values give the color, while the alpha fixes the opacity.

For this project (and actually it is the case of colors on the web), these values are integers between 0 and 255 . The value of one color represent the quantity of light of that color being added to the mix; the final color is produced according to additive color mixing. The alpha represents the opacity: an alpha of 0 means complete transparency, an alpha of 255 means solid color.

For example:

- A pixel with value (255,255,255,255) has all RGB colors to the max, and is a white pixel (full light).
- A pixel with value (0,0,0,255) has all RGB colors to 0, and is a black pixel (no light).
- A pixel with value (255,0,0,255) only has its red component to the max and no green or blue, so it is a red pixel.
- A pixel with value (0,255,0,255) only has its green component to the max and no green or blue, so it is a green pixel.
- A pixel with value (0,0,255,255) only has its blue component to the max and no green or blue, so it is a blue pixel.
- A pixel with value (127,127,127,255) has all three components to a middle value, so it is a gray pixel.
- A pixel with value (255,255,0,255) has both red and green components to the max but no blue, so it a yellow pixel.
- A pixel with value (255,0,255,255) has both red and blue components to the max but no green, so it a magenta pixel.
- A pixel with value (0,255,255,255) has both green and blue components to the max but no red, so it a cyan pixel.
- A pixel with value (127,0,0,255) only has its red component to a middle and no green or blue, so it is a dark red pixel.

to have a better sense of what these value means, it is possible to experiment using an HTML color picker.

Note: the alpha is often displayed as a value α between 0 and 1, but it is still stored as a value A between 0 and 255. The conversion is as follows: $\alpha = \frac{A}{255}$ and $A = \lfloor 255 \times \alpha \rfloor$.

P3.2 Code preamble and first manipulation: input/output

The project uses the `PIL` library (PIL stands for *Python Imaging Library*), that is found in the package `pillow`. It may require installation, either by running

```
pip3 install -U pillow
```

in a terminal or the following in Jupyter:

```
!pip3 install -U pillow
```

To actually use the package, the first thing to do is to import the `Image` module from the `PIL` library:

```
from PIL import Image
```

The `open` function in the `Image` module imports an image file. This function requires the path to the image, which can be relative to the place where this script is stored or absolute in the filesystem. For example, here, the file `Wrench.png` located in the subdirectory `images` is being imported; this directory is located in the same directory as the script.

```
wrench = Image.open("images/Wrench.png")
```

The picture can then be displayed using `show()`:

```
wrench.show()
```

It should open up a window to display this image:

Depending on the system, the transparency of the picture may or may not be adequately represented (here there is no background). So `show()` might be good for debugging, but to see the picture better, it might be better to open it through the file browser. If the picture has been changed (as will happen throughout the project), it needs to be saved first:

```
wrench.save("/tmp/WrenchCopy.png")
```

Here, an absolute path is given as argument: it starts with a `/`. So the file is being saved in the `/tmp/` folder at the root of the file system (in GNU/Linux).

Note: in Windows, the temporary folder is `C:\Users\<username>\AppData\Local\Temp`, where `<username>` is the current user logged into the system.

P3.3 Image format

P3.3.1 Pixel organization

The PNG image format used in this project stores images as two-dimensional array of pixels. The origin $(0,0)$ is located at the top left corner of the image. The x coordinate grows horizontally from left to right (so the usual). The y coordinate grows vertically but from top to bottom (so not so usual). The size of the image is called the *resolution*, and is often written as the product $X \times Y$ where X is the width and Y is the height, in pixels.

$0,0$	$1,0$	$\cdots$	$X-1,0$
$0,1$	$1,1$	$\cdots$	$X-1,1$
$0,2$	$1,2$	$\cdots$	$X-1,2$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$0,Y-1$	$1,Y-1$	$\cdots$	$X-1,Y-1$

This array of cells can be reached through the `load()` command. Then an individual pixel is reached by its coordinates. But in order to know where these coordinates end, the resolution of the image must be known, which is conveniently done by the `size` command.

```
xSize,ySize = wrench.size                  # Get the size in variables␣
↪xSize,ySize
pixWrench = wrench.load()                  # Get the array of pixels
pixCenter = pixWrench[xSize//2,ySize//2]   # Get a pixel in the middle
print("One pixel in the middle:",pixCenter)
```

For a refresher on the two forms of division, see Section L3.4.2.

```
One pixel in the middle: (67, 79, 93, 255)
```

Each pixel is a *tuple* of 4 values that represent the amount of Red, Green, Blue, and Alpha (RGBA) in the pixel.

Exercise

Uncomment the following line and try to run it.

```
#pxError = pixWrench[xSize,ySize]           # This gives an error
```

Explain why an error was produced.

P3.3.2 Creating an Alpha channel

Not all images actually come with pixels in 4 dimensions: in some cases the Alpha value is missing. To make sure that images are in the RGBA mode and can be used in the functions, right after opening, the following `addalpha` function should be called on the image.

```
def addalpha(image):
    if image.mode != "RGBA":
        image.putalpha(255)
    return image

statueLiberty = Image.open("images/Liberty.png") # RGB mode
 # print a pixel in the middle: only 3 components
print("Before addalpha:",statueLiberty.load()[statueLiberty.size[0]//
 ↪2,statueLiberty.size[1]//2])
addalpha(statueLiberty)                                    # Now ready to be used
 # print a pixel in the middle: now there is an alpha component
print("After addalpha: ",statueLiberty.load()[statueLiberty.size[0]//
 ↪2,statueLiberty.size[1]//2])
```

```
Before addalpha: (69, 94, 90)
After addalpha:  (69, 94, 90, 255)
```

P3.3.3 Experiments with resolution and colors

One way to experiment with both resolution and color is by creating new pictures filled with a given color, using the `new()` function in the `Image` module.

The syntax is as follows: `res = Image.new("RGBA",(x,y),(red,green,blue,alpha))` `RGBA` indicates we are using this model for colors, `x` and `y` provide the resolution, and `(red,green,blue,alpha)` give the value of the color that will be in every pixel.

```
orange = Image.new("RGBA",(123,45),(255,153,0,255))
orange.show()
```

Exercise

Experiment with the `new` function by creating at least three images of your liking. For example:

- A pink square of size 130 pixels
- A dark blue rectangle of resolution 256×128
- A light-gray rectangle of resolution 512×724

Remember that it is easier to find the RGB value of colors using an HTML color picker.

```
#
# Code here
#
```

P3.4 Changing images pixel by pixel

In order to *change* the image, each pixel is accessed individually and modified accordingly. Note that since the image is being changed, it is wise to copy the image before testing; otherwise several effects will be applied to the same image. This is done by the `copy()` function:

```
wrench2 = wrench.copy()
```

P3.4.1 Coloring the image by adding light

The first kind of simple manipulation of this project is to change the color of the image by accenting one component of each pixel. For example, the function below changes the image by setting the blue component of all pixels to the maximum value 255.

```
def bluize(image):
    px = image.load()          # Get the array of pixels
    xsize,ysize = image.size   # Get the resolution
    for x in range(0,xsize):
        for y in range(0,ysize): # Go through the whole array
            px[x,y] = (px[x,y][0],px[x,y][1],255,px[x,y][3])
                # Set the blue component to 255, the rest stays the same
    return image

# Testing
blueWrench = wrench.copy() # Make a copy because the image is being
 ↪modified
bluize(blueWrench)
blueWrench.show()
#wrench.show()           # Uncomment to see there was no change done to
 ↪the original picture
```

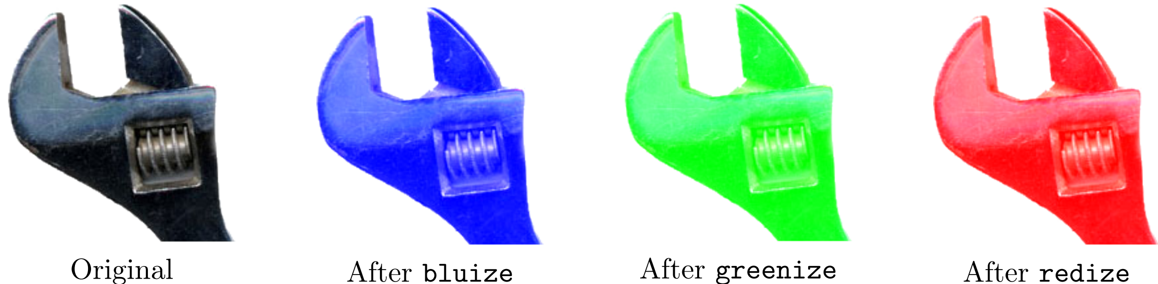
For a refresher on loops, see Lecture L6.

Exercise: Green and red versions

Code and test the functions `greenize` and `redize` that color the image in a similar fashion: `greenize` makes it greener by boosting the green component, and `redize` makes it redder by boosting the red component.

```
#
# Code here
#
```

Here are the colored versions of the wrench image:



Original After `bluize` After `greenize` After `redize`

Remark that since light was added to one component, the image seems brighter than the original.

P3.4.2 Coloring the image by removing light

As the power of a component was increased, it is also possible to decrease it to 0. For example, if the blue component is removed from a picture, it becomes darker and more yellow.

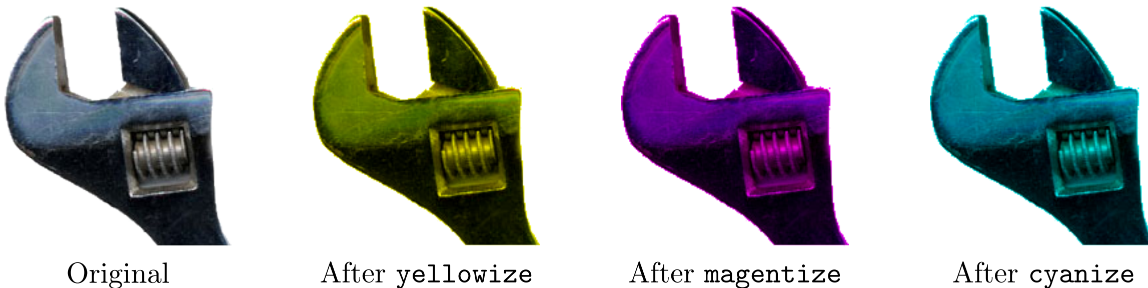
```
def yellowize(image):
    px = image.load()           # Get the array of pixels
    xsize,ysize = image.size    # Get the resolution
    for x in range(0,xsize):
        for y in range(0,ysize): # Go through the whole array
            px[x,y] = (px[x,y][0],px[x,y][1],0,px[x,y][3])
                    # Set the blue component to 0, the rest stays the same
    return image

# Testing
yellowWrench = wrench.copy() # Make a copy because the image is being␣
 ↪modified
yellowize(yellowWrench)
yellowWrench.show()
```

Exercise: Magenta and cyan versions

Code and test the functions `magentize` and `cyanize` that color the image in a similar fashion: `magentize` makes it magenta by removing the green component, and `cyanize` makes it cyan by removing the red component.

```
#
# Code here
#
```

Here are the colored versions of the wrench image:



Original After `yellowize` After `magentize` After `cyanize`

P3.4.3 Other color manipulations

Exercise: Negative image

Code and test the function `negate` that changes the image to its negative image. The negative image is obtained by changing each color component from x to its complement to 255: $255 - x$. Note that the opacity remains the same.

```
#
# Code here
#
```

Exercise: Grayscale

Code and test a function `grayscale` that converts it to a grayscale image. A grayscale image has only grey pixels: pixels that have all RGB components to the same value. This value is the average of the RGB components, because it corresponds to the total quantity of light that this pixel has. For example a pixel that originally has the value (158,66,245,255) (purple) will become (156,156,156,255), because $\left\lfloor \frac{158+66+245}{3} \right\rfloor = 156$.

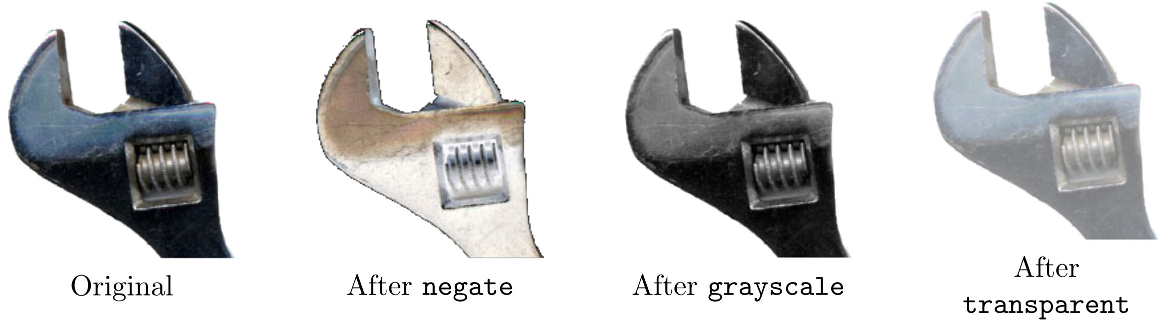
```
#
# Code here
#
```

Exercise: Transparency

Code and test a function `transparent` that halves the alpha value of each pixel in order to make the image semi-transparent.

```
#
# Code here
#
```

Here is an example of the above three manipulations on the wrench image:



Original | After `negate` | After `grayscale` | After `transparent`

P3.5 Global and vicinal manipulation

Up to now, all functions manipulate each pixel by itself, without looking at other pixels in the image in order to change the pixel value.

P3.5.1 Global brightness of an image

One interesting measure about an image in its globality is how bright or dark it is. This is done by averaging the brightness of each pixel; a pixel's brightness being the average of its RGB components (rounded to the lowest integer).

```
def brightness(image):
    res = 0
    px = image.load()
    xsize,ysize = image.size                  # Get the resolution
    for x in range(0,xsize):
        for y in range(0,ysize):              # Go through the whole array
                for c in range(0,3):          # Go through each RGB component
                    res = res+px[x,y][c]      # Add all values to res
    res = res // (3*xsize*ysize)              # Divide by the number of values we added: 3 per pixel
    return res

print("Brightness of the 'Wrench.png' image:",brightness(wrench))
```

```
Brightness of the 'Wrench.png' image: 150
```

Exercise: A better brightness measure

Although this is an interesting function, it does not take into account the fact that some pixels are actually not visible, or just less visible, if their alpha is not 255.

Code and test a function `realBrightness` that takes that into account by weighing the luminosity of each pixel by the alpha value of the pixel.

Note: When doing a weighted average, the sum of weighted values must be divided by the sum of the weights. Mathematically, the average of values $(a_1, a_2, \dots, a_n)$ with respective weights $(w_1, w_2, \dots, w_n)$ is

$$\text{Avg} = \frac{\sum_{i=1}^{n} a_i \times w_i}{\sum_{i=1}^{n} w_i} = \frac{a_1 \times w_1 + a_2 \times w_2 + \cdots + a_n \times w_n}{w_1 + w_2 + \cdots + w_n}$$

```
#
# Code here
#
```

```
Real brightness of the 'Wrench.png' image: 71
```

Exercise: A black-and-white image based on the brightness

The result of the brightness function can be used to create a black-and-white version of the image. Do not confuse black and white with grayscale: in a black and white picture the only two possible RGB values for a pixel are 0,0,0 (black) or 255,255,255 (white). The alpha value remains the same as before, however.

To determine whether a pixel should be black or white, the brightness is used as a *threshold*: if the pixel is brighter than the global image brightness, then the pixel is white, otherwise it is black.

Code and test the `threshold` function that makes the image black-and-white according to the *real brightness* of the image.

```
#
# Code here
#
```

P3.5.2 Blurring an image

To blur an image, the color of a pixel (x, y) is determined using the pixels around it (its "neighbor pixels"). Namely, the color of a pixel is the average of the color of the 9 pixels surrounding (x, y) (including the pixel itself).

$x-2, y-2$	$x-1, y-2$	$x, y-2$	$x+1, y-2$	$x+2, y-2$
$x-2, y-1$	$x-1, y-1$	$x, y-1$	$x+1, y-1$	$x+2, y-1$
$x-2, y$	$x-1, y$	x, y	$x+1, y$	$x+2, y$
$x-2, y+1$	$x-1, y+1$	$x, y+1$	$x+1, y+1$	$x+2, y+1$
$x-2, y+2$	$x-1, y+2$	$x, y+2$	$x+1, y+2$	$x+2, y+2$

Things to consider in both functions below

- The color is averaged component by component, alpha included.
- Near the border and corners, all neighbors may not actually exist, and that must be checked by the function
- Because the old value of a pixel is also used when calculating the average for his neighbors, the function must keep a copy of the original image as a source.

Exercise: Blurring with fixed neighborhood

Code and test a function `blur` that blurs the image using the above method.

```
#
# Code here
#
```

Exercise: Blurring with variable neighborhood

The blurring of the above function is actually quite hard to see. A stronger blurring effect can be achieved by using a larger neighborhood.

Code and test a function `blurDist` that takes two parameters: the image, and a distance. The distance is used to determine what is the neighborhood being considered, by giving how many steps left, right, above, and below are in the neighborhood.

For example, if distance=1, it boils down to the previous case. If distance=k, the area to consider is depicted in Figure P3.1.

```
#
# Code here
#
```

Here is an example of the above three manipulations on the wrench image:

Original | After `threshold` | After `blur` | After `blurDist` with distance=10

P3.6 Doing all transformations onto files

In this project, several transformations were programmed (12 in total). It is practical to generate them all in one go for a given input file. The input file is given as a *path* to a picture. For the output files, only a base path and name is given, and an appropriate suffix (including `.png` file extension) should be added by the program. The choice of suffix is left to the programmer.

For example if the path to the picture is `images/Wrench.png`, and the path to the basename is `images/generated/Wrench_` (the folder has to exist). The “bluized” version could use the suffix `Blue`, so the saving path for this transformed image would be `images/generated/Wrench_Blue.png`.

For example consider this function `convert2grayscale` that takes an input path and an output basename:

```python
def convert2grayscale(inpath, outbase):
    img = Image.open(inpath)
    addalpha(img)
    # Grayscale(img)  # uncomment this line when grayscale has been defined
    img.save(outbase+"Grayscale.png")
    return

convert2grayscale("images/Wrench.png","/tmp/Wrench")
```

$x-(k+1), y-(k+1)$	$x-k, y-(k+1)$	$\cdots$	$x-1, y-(k+1)$	$x, y-(k+1)$	$x+1, y-(k+1)$	$\cdots$	$x+k, y-(k+1)$	$x+(k+1), y-(k+1)$
$x-(k+1), y-k$	$x-k, y-k$	$\cdots$	$x-1, y-k$	$x, y-k$	$x+1, y-k$	$\cdots$	$x+k, y-k$	$x+(k+1), y-k$
$\vdots$	$\vdots$	$\ddots$	$\cdots$	$\cdots$	$\cdots$	$\iddots$	$\vdots$	$\vdots$
$x-(k+1), y-1$	$x-k, y-1$	$\cdots$	$x-1, y-1$	$x, y-1$	$x+1, y-1$	$\cdots$	$x+k, y-1$	$x+(k+1), y-1$
$x-(k+1), y$	$x-k, y$	$\cdots$	$x-1, y$	x, y	$x+1, y$	$\cdots$	$x+k, y$	$x+(k+1), y$
$x-(k+1), y+1$	$x-k, y+1$	$\cdots$	$x-1, y+1$	$x, y+1$	$x+1, y+1$	$\cdots$	$x+k, y+1$	$x+(k+1), y+1$
$\vdots$	$\vdots$	$\iddots$	$\cdots$	$\cdots$	$\cdots$	$\ddots$	$\vdots$	$\vdots$
$x-(k+1), y+k$	$x-k, y+k$	$\cdots$	$x-1, y+k$	$x, y+k$	$x+1, y+k$	$\cdots$	$x+k, y+k$	$x+(k+1), y+k$
$x-(k+1), y+(k+1)$	$x-k, y+(k+1)$	$\cdots$	$x-1, y+(k+1)$	$x, y+(k+1)$	$x+1, y+(k+1)$	$\cdots$	$x+k, y+(k+1)$	$x+(k+1), y+(k+1)$

Figure P3.1: The red cells are not taken into account when calculating the color of pixel x, y with distance k.

P3.6.1 Exercise: The wrap-up function

Code and test a function `generateAll` that creates the 12 versions of the images and save them. Recall that these are all the versions:

- blue coloring
- green coloring
- red coloring
- yellow coloring
- magenta coloring
- cyan coloring
- negative
- grayscale
- transparent
- black-and-white
- blurred (fixed neighborhood)
- blurred (parameterized neighborhood)

It can be tried on more PNG images of your choice!

```
#
# Code here
#
```

Project 4
Simple Operations with Fractions

Project contents

P4.1 Introduction **109**
P4.1.1 Simplifying or reducing fractions: the procedure 109
P4.2 The Greatest Common Divisor (GCD) **109**
P4.2.1 The brute force approach 110
P4.2.2 Euclid's Algorithm 111
P4.3 Reducing fractions **113**
P4.4 Conversion to decimal value **114**
P4.5 The Four Operations with Fractions (Exercise) **116**

P4.1 Introduction

The goal of this project is to write a program able to:

- read and write fractions
- simplify fractions
- add, subtract, multiply, and divide fractions

P4.1.1 Simplifying or reducing fractions: the procedure

A fraction of integers $\frac{a}{b}$ can be seen as the ordered pair (a, b). For example $\frac{6}{9}$ is represented by $(6, 9)$.

In order to *simplify* (or *reduce*) this fraction, one has to:

1. Find the Greatest Common Divisor (GCD) of 6 and 9; it is 3.
2. Divide each number in the pair by the GCD: $6 \div 3 = 2$ and $9 \div 3 = 3$
3. The result yields the par $(2, 3)$ which represents the fraction $\frac{2}{3}$.

Now $\frac{2}{3}$ is irreducible, while $\frac{6}{9}$ is reducible.

P4.2 The Greatest Common Divisor (GCD)

The first step part of the simplification procedure is to find the value of the GCD.

P4.2.1 The brute force approach

In the *brute force approach*, all numbers between 1 and the lowest between a and b are tried. A number k is a common divisor of a and b if it divides them both, meaning the remainder of the division by k is 0.

This procedure can be translated into a `gcd` function, which takes two parameters: `a` and `b`.

To define a function in Python, it needs a name and a list of parameters.

For a more thorough introduction on functions, see Lecture L8.

```
def somefunction():         # No parameters
    line 1
    line 2
    ...

def another function(a,b):  # This has 2 parameters
    line 1
    line 2
    ...
```

The name of the function cannot be a reserved word such as: `if`, `elif`, `else`, `for`, `while`, `True`, `False`.

```
def gcd(a,b):
    g = 1  # 1 divides all numbers
    for k in range(2,min(a,b)+1):
        if a%k == 0 and b%k == 0: # Does k divide a? and does k divide b?
            g = k                 # It is a common divisor, update g
    return g

def main():
    while True:              # Continue until 'q' is entered
        ans = input('Press q to quit or a pair of integers: ')
        if ans == 'q':
            break
        ans = ans.split(' ') # Separate the input in two integers
        a = int(ans[0])
        b = int(ans[1])
        g = gcd(a,b)         # Call the gcd function
        print('gcd({},{}) = {}'.format(a,b,g))

main() # Call the main function
```

```
Press q to quit or a pair of integers: 42 27
gcd(42,27) = 3
Press q to quit or a pair of integers: 32 8
gcd(32,8) = 8
Press q to quit or a pair of integers: 12 35
gcd(12,35) = 1
Press q to quit or a pair of integers: q
```

This is actually very inefficient: there are $2 \times (\min(a, b) - 1)$ divisions performed. So for example `gcd(10000, 10001)` requires approximately 20000 divisions. But it can actually be done using only 3!

P4.2.2 Euclid's Algorithm

Euclid noticed that `gcd(a,b)` is `gcd(b,a%b)`. This can be turned into a program that requires in far fewer `%` operations.

```
def gcd(a,b):
    r0 = a
    r1 = b
    while r1 != 0:
        r2 = r0 % r1
        r0 = r1
        r1 = r2
    return r0

def main():
    while True:                 # Continue until 'q' is entered
        ans = input('Press q to quit or a pair of integers: ')
        if ans == 'q':
            break
        ans = ans.split(' ') # Separate the input in two integers
        a = int(ans[0])
        b = int(ans[1])
        g = gcd(a,b)            # Call the gcd function
        print('gcd({},{}) = {}'.format(a,b,g))

main() # Call the main function
```

```
Press q to quit or a pair of integers: 6 9
gcd(6,9) = 3
Press q to quit or a pair of integers: 6 6
gcd(6,6) = 6
Press q to quit or a pair of integers: 12 3
gcd(12,3) = 3
Press q to quit or a pair of integers: 47 12
gcd(47,12) = 1
Press q to quit or a pair of integers: q
```

Evaluating the efficiency

How much more effective is Euclid's version? This can be tested by counting the number of divisions performed.

```
def gcdcount(a,b):
    r0 = a
    r1 = b
```

```
    k = 0
    while r1 != 0:
        r2 = r0 % r1
        k += 1 # Count the number of times this loop is executed
        r0 = r1
        r1 = r2
    return k   # k is returned, not the GCD

def main():
    while True:
        ans = input('Press q to quit or a pair of integers: ')
        if ans == 'q':
            break
        ans = ans.split(' ') # Separate the input in two integers
        a = int(ans[0])
        b = int(ans[1])
        k = gcdcount(a,b)    # Call the gcdcount function
        print('gcd({},{}) takes {} modulo operations'.format(a,b,k))

main() # Call the main function
```

```
Press q to quit or a pair of integers: 6 6
gcd(6,6) takes 1 modulo operations
Press q to quit or a pair of integers: 6 9
gcd(6,9) takes 3 modulo operations
Press q to quit or a pair of integers: 1000 1001
gcd(1000,1001) takes 3 modulo operations
Press q to quit or a pair of integers: 1000000000000 1000000000001
gcd(1000000000000,1000000000001) takes 3 modulo operations
Press q to quit or a pair of integers: 47 12
gcd(47,12) takes 3 modulo operations
Press q to quit or a pair of integers: 1597 987
gcd(1597,987) takes 15 modulo operations
Press q to quit or a pair of integers: q
```

The efficiency can be evaluated in a more systematic way by trying to calculate the GCD of all pairs of numbers below a bound.

```
def gcdcount(a,b):
    r0 = a
    r1 = b
    k = 0
    while r1 != 0:
        r2 = r0 % r1
        k += 1 # Count the number of times this loop is executed
        r0 = r1
        r1 = r2
    return k   # k is returned, not the GCD
```

```
def main():
    while True:
        ans = input('Press q to quit or a integer: ')
        if ans == 'q':
            break
        a = int(ans)
        m = 1
        for i in range (1,a+1):     # Try all pairs of numbers (i,j) with␣
↪1≤i≤j≤a
            for j in range(i,a+1):
                k = gcdcount(i,j)
                if m < k:
                    m = k
        print('The GCD of any pair or numbers in 1..{} takes at most {}␣
↪modulo operations.'.format(a,m))
main() # Call the main function
```

```
Press q to quit or a integer: 10
The GCD of any pair or numbers in 1..10 takes at most 5 modulo operations.
Press q to quit or a integer: 100
The GCD of any pair or numbers in 1..100 takes at most 10 modulo␣
↪operations.
Press q to quit or a integer: 1000
The GCD of any pair or numbers in 1..1000 takes at most 15 modulo␣
↪operations.
Press q to quit or a integer: 10000
The GCD of any pair or numbers in 1..10000 takes at most 19 modulo␣
↪operations.
Press q to quit or a integer: q
```

P4.3 Reducing fractions

Fractions can be simplified using the `gcd` function that was previously written.

Each fraction is inputted as string in the form `a/b`. This string can be split into a list of possibly 2 strings which are separated by the `/` character. If the input is just an integer `a`, then it is as if the fraction was `a/1`, which is already irreducible. The result of the fraction reduction is also a string.

```
def reduce(f):
    f = f.split('/') # Split the input.
    num = int(f[0])  # Numerator
    if len(f) == 1:  # Fraction 'a' is like a/1, already reduced
        return f[0]
    den = int(f[1])  # Denominator
    g = gcd(num,den) # Use the gcd function
    num = num//g     # Divide (quotient) both numerator
```

```
    den = den//g      # and denominator by the GCD
    if den < 0:       # No negative sign as the denominator
        num *= -1
        den *= -1
    if den == 1:      # Case a/1 (after simplification)
        f = str(num)
    else:
        f = str(num) + '/' + str(den)
    return f

def main():
    while True:
        f = input('Press q to quit or a fraction or integer to continue: ')
        if f == 'q':
            break
        fred = reduce(f)
        print('{} = {}'.format(f,fred))

main()
```

```
Press q to quit or a fraction or integer to continue: 6/9
6/9 = 2/3
Press q to quit or a fraction or integer to continue: -6/-9
-6/-9 = 2/3
Press q to quit or a fraction or integer to continue: -6/9
-6/9 = -2/3
Press q to quit or a fraction or integer to continue: 6/-9
6/-9 = -2/3
Press q to quit or a fraction or integer to continue: -8/-4
-8/-4 = 2
Press q to quit or a fraction or integer to continue: -6/27
-6/27 = -2/9
Press q to quit or a fraction or integer to continue: 6/-27
6/-27 = -2/9
Press q to quit or a fraction or integer to continue: 27/-6
27/-6 = -9/2
Press q to quit or a fraction or integer to continue: -27/6
-27/6 = -9/2
Press q to quit or a fraction or integer to continue: q
```

P4.4 Conversion to decimal value

As before, each fraction is entered as string a/b or a (which is the case with integers). Real division is used here to produce a `float`.

```
def decimal(f):
    f = f.split('/')
    if len(f) == 2:
        num = int(f[0])
        den = int(f[1])
    elif len(f) == 1:
        num = int(f[0])
        den = 1
    else:
        return 'This is not valid fraction.'
    return num/den

def main():
    while True:
        ans = input('Press q to quit or a fraction or integer to continue:
 ↪ ')
        if ans == 'q':
            break
        f = reduce(ans)
        fd = decimal(f)
        print( '{} = {} = {}'.format(ans,f,fd) )

main()
```

```
Press q to quit or a fraction or integer to continue: 4/8
4/8 = 1/2 = 0.5
Press q to quit or a fraction or integer to continue: -5/20
-5/20 = -5/20 = -0.25
Press q to quit or a fraction or integer to continue: 89
89 = 89 = 89.0
Press q to quit or a fraction or integer to continue: 20/5
20/5 = 4 = 4.0
Press q to quit or a fraction or integer to continue: q
```

Since the numerator and the denominator of fractions are frequently needed, the splitting code is repeated at several places. Instead, it is cleaner to write two new functions: - a function `num` that returns the value of the numerator; - a function `den` that returns the value of the denominator.

The `reduce` and `decimal` functions can be rewritten using these new functions.

```
def num(f):
    f = f.split('/')
    return int(f[0])

def den(f):
    f = f.split('/')
    if len(f) == 1:
        return 1
```

```
    else:
        return int(f[1])

def reduce(f):
    n = num(f) # Call to the new function num
    d = den(f) # Call to the new function den
    # Regularization of the fraction (as before)
    g = gcd(n,d)
    n = n//g
    d = d//g
    if d < 0:
        n *= -1
        d *= -1
    if d == 1:
        return str(n)
    else:
        return str(n)+'/'+str(d)

def decimal(f):
    return num(f)/den(f)

def main():
    while True:
        ans = input('Press q to quit or a fraction or integer to continue:
 ↪ ')
        if ans == 'q':
            break
        f = reduce(ans)
        fd = decimal(f)
        print('{} = {} = {}'.format(ans,f,fd))

main()
```

```
Press q to quit or a fraction or integer to continue: 1/3
1/3 = 1/3 = 0.3333333333333333
Press q to quit or a fraction or integer to continue: 2/-6
2/-6 = -1/3 = -0.3333333333333333
Press q to quit or a fraction or integer to continue: -6/-2
-6/-2 = 3 = 3.0
Press q to quit or a fraction or integer to continue: -23
-23 = -23 = -23.0
Press q to quit or a fraction or integer to continue: q
```

P4.5 The Four Operations with Fractions (Exercise)

Fractions can be added, subtracted, multiplied, and divided.

The addition function `add` is provided. Based on this, code the subtraction, multiplication, and division functions below.

```
def add(f1,f2):
    n1 = num(f1)
    d1 = den(f1)
    n2 = num(f2)
    d2 = den(f2)
    n = n1*d2+d1*n2                   # Numerator of the sum
    d = d1*d2                         # Denominator of the sum
    f = reduce(str(n) + '/' + str(d)) # Reduce the result
    return f

#
# Define a function for subtraction here
#

#
# Define a function for multiplication here
#

#
# Define a function for division here
#

def main():
    while True:
        ans = input('Press q to quit or a pair of fractions or integers␣
 ↪to continue: ')
        if ans == 'q':
            break
        ans = ans.split(' ') # Separate the two operands from the input
        f = reduce(ans[0])
        g = reduce(ans[1])
        # Addition
        res = add(f,g)
        print('{} + {} = {}'.format(f,g,res))
        fd = decimal(f)
        gd = decimal(g)
        resd = decimal(res)
        print('{} + {} = {}'.format(fd,gd,resd))
        # Subtraction
        #
        # Complete the code here
        #
        # Multiplication
        #
        # Complete the code here
        #
        # Division
```

```
        #
        # Complete the code here
        #

main()
```

Project 5
Time Measurement and Dates

Project contents

P5.1 Introduction **119**
P5.1.1 Library functions 119
P5.2 Letting time elapse **120**
P5.2.1 Exercise 120
P5.3 An aside: erasing characters **120**
P5.3.1 Exercise 122
P5.4 A cooking timer **122**
P5.4.1 Exercise 123
P5.5 Current date and time **123**
P5.5.1 Exercise 126
P5.6 Measuring time **127**
P5.6.1 Exercise 127
P5.6.2 Exercise 128
P5.6.3 Exercise 128

P5.1 Introduction

The goal of this project is to manipulate representations of *time* in computers. It will be used to create timers, countdowns, and display the current date and time.

P5.1.1 Library functions

The `time` library needs to be imported. Only two functions from this library will be used.

- `time.time()` that gives the current time in seconds (with decimal point); more explanations about what that value is exactly in Section *Current date and time*.
- `time.sleep(x)` that pauses the execution of the code for `x` seconds.

```
import time

print(time.time())  # current time
time.sleep(5)       # wait 5 seconds
print(time.time())  # current time
```

```
1595887384.8185391
1595887389.8236763
```

As can be seen from the above, `time.sleep` is not 100% accurate, but that will do.

P5.2 Letting time elapse

By using `time.sleep`, one can have the computer wait. In the example below, each letter is printed individually.

```
def slowtyping(word,wait=1):
    n = len(word)                # length of the string
    for i in range(0,n):         # repeat for i in 0, 1, ..., n-1
        print(word[i],end="") # print character at position i in the␣
↪string
        time.sleep(wait)         # wait some time

slowtyping("Hello World!",0.25)
```

```
Hello World!
```

This can also be used to count down:

```
def countDown(dur,endSentence):
    for k in range(dur,0,-1): # repeats for k in dur, dur-1, ..., 2, 1
        print(k,end="... ")
        time.sleep(1)
    print(endSentence)
    return

countDown(10,"Happy New Year!")
```

Throughout this project, the code must be run to see interactivity!

```
10... 9... 8... 7... 6... 5... 4... 3... 2... 1... Happy New Year!
```

P5.2.1 Exercise

Write a function `countTo(n)` that counts from 1 to `n`, waiting 1 second between each number

```
#
# Code here
#
```

P5.3 An aside: erasing characters

Since there are lots of things to display, it is useful to *replace* what was written instead of writing after. In order to do that, the **backspace** character `\b` can be used. Note: The backspace character behaves a bit differently in a terminal and in Jupyter

- In a terminal, the backspace character moves the position of the writing cursor one character to the left. It does not actually erase anything.
- In Jupyter, it actually erases, but there are bugs when several `\b` are printed together, so they have to be printed one by one, and the print should *flush* the output.

```
def writeAndErase(msg,wait=1,initialextrawait=3): # This works in Jupyter␣
↪only
    n = len(msg)
    print(msg,sep="",end="")  # prints the message
    time.sleep(initialextrawait)
    for i in range(0,n):
        time.sleep(wait)
        print("\b",end="",flush=True)
    return

writeAndErase("Hello world!",wait=0.25,initialextrawait=2)
```

```
def writeAndErase(msg,wait=1,initialextrawait=3): # This works in terminal
    n=len(msg)
    backspaces="\b"*n
    print(msg,sep="",end="",flush=True)  # Print the message
    time.sleep(initialextrawait)
    for i in range(0,n):
        time.sleep(wait)
        print("\b \b",sep="",end="",flush=True) # Move cursor back, print␣
↪whitespace, move cursor back again
    print()
    return
```

Using this technique, a countdown that always erases the previous number can be coded:

```
def countDownWithErasing(dur,endSentence):  # This works in Jupyter only
    print(" ",end="")
    prevLength = 0                  # Number of characters to erase
    for k in range(dur,0,-1):
        for j in range(0,prevLength):
            print("\b",end="",flush=True)     # Erase the characters
        msg = str(k)                # The number as a string
        prevLength = len(msg)       # Number of characters we will have to␣
↪erase next time
        print(msg,end="")           # Print the number
        time.sleep(1)               # Wait 1 second
    for j in range(0,prevLength):
        print("\b",end="")          # Erase the characters
    print(endSentence)
    return

countDownWithErasing(10, "Happy New Year!")
```

```
def countDownWithErasing(dur,endSentence): # This works in terminal only
    prevLength=0                        # Number of characters to erase
    for k in range(dur,0,-1):
```

```
        backspaces="\b"*prevLength # Move the cursor back
        whitespaces=" "*prevLength # The same amount of whitespaces
        msg = str(k)               # The number as a string
        prevLength = len(msg)      # Number of characters we will have to erase next time
        print(backspaces,whitespaces,backspaces,msg,sep="",end="",flush=True)
        time.sleep(1)
    backspaces+="\b"*prevLength
    print(backspaces,endSentence,sep="\t",flush=True)
    time.sleep(1)
    return
```

```
Happy New Year!
```

P5.3.1 Exercise

Write a `countToWithErasing(n)` function that counts from 1 to `n`, erasing the previous number each time.

```
#
# Code here
#
```

P5.4 A cooking timer

To count down 3 minutes, one could simply count down $3 \times 60 = 180$ seconds. But that is not very practical, and it gets worse if the user wants to measure 2 hours...

The goal here is to make a better timer, that takes minutes and seconds. The principle is as follows: once the seconds are down to 0, decrease the minute and set the seconds to 59.

```
def eggTimer(minutes,seconds):
    print(" ",end="")
    prevLength = 0 # number of characters to erase
    for m in range(minutes,-1,-1):
        for s in range(seconds,-1,-1):
            for j in range(0,prevLength):
                print("\b",end="",flush=True)  # Erase the characters
            #msg = str(m)+" minutes "+str(s)+" seconds" # With conversion to string
            msg = "%d minutes %d seconds" %(m,s)      # With formatted string
            prevLength = len(msg)
            print(msg,end="")
            time.sleep(1)
        seconds=59 # reset the seconds to 59 for next minute
    for j in range(0,prevLength):
        print("\b",end="",flush=True)  # Erase the characters
```

```
    print("It's ready!")
    return

eggTimer(1,3) # 1 minute and 3 seconds
```

```
def eggTimer(minutes,seconds): # Works in terminal only
    print(" ",end="")
    prevLength = 0 # number of characters to erase
    for m in range(minutes,-1,-1):
        for s in range(seconds,-1,-1):
            print("\b"*prevLength," 
 ↪"*prevLength,"\b"*prevLength,sep="",end="",flush=True)  # Erase the 
 ↪characters
            #msg = str(m)+" minutes "+str(s)+" seconds" # With conversion 
 ↪to string
            msg = "%d minutes %d seconds" %(m,s)       # With formatted 
 ↪string
            prevLength = len(msg)
            print(msg,end="",flush=True)
            time.sleep(1)
        seconds=59 # reset the seconds to 59 for next minute
    print("\b"*prevLength," 
 ↪"*prevLength,"\b"*prevLength,sep="",end="",flush=True)  # Erase the 
 ↪characters
    print("It's ready!")
    return

eggTimer(1,3) # 1 minute and 3 seconds
```

```
It's ready!
```

P5.4.1 Exercise

Write a cooking timer that takes hours, minutes, and seconds.

```
#
# Code here
#
```

```
It's ready!
```

P5.5 Current date and time

The number of seconds returned by `time.time()` is actually the number of seconds since January 1st, 1970 at 00:00 in GMT timezone (this particular time is called *epoch*).

This number of seconds can be converted to an actual date and time. For the time, the remainder of the Euclidean division by 60 gives the seconds, the quotient by 60 being the

number of minutes (since epoch). Taking the remainder of this by 60 gives the minutes, while the quotient gives the number of hours, etc.

```
def readableTime(zone):
    t = int(time.time())    # t is the number of seconds since epoch (no␣
↪decimal point)
    seconds = t%60
    t = t//60               # t is now the number of minutes since epoch
    minutes = t%60
    t = t//60               # t is now the number of hours since epoch
    t = t+zone              # add the Time Zone difference from GMT
    hours = t%24
    t = t//24               # t is now the number of days since epoch
    res = str(hours)+":"+str(minutes)+":"+str(seconds)
    return res

readableTime(-4) # Eastern Summer Time
```

```
'16:33:22'
```

To convert the number of days into an actual date, there is no formula that gives the exact value, mainly because months have different length and there are leap years.

The condition for leap years is as follows: leap years are multiple of 4 except years multiple of 100 except years multiples of 400. For example: 2019 was not a leap year (not multiple of 4), 2020 is a leap year (multiple of 4 but not of 100), 2100 will not be (multiple of 100 but not 400), but 2000 was (multiple of 400).

This can be translated into the following function:

```
def isLeap(y):
    if (y%400==0):
        return True
    elif (y%100==0):
        return False
    elif (y%4==0):
        return True
    else:
        return False

print("Is 2019 a leap year?",isLeap(2019))
print("Is 2020 a leap year?",isLeap(2020))
print("Is 2100 a leap year?",isLeap(2100))
print("Is 2000 a leap year?",isLeap(2000))
```

```
Is 2019 a leap year? False
Is 2020 a leap year? True
Is 2100 a leap year? False
Is 2000 a leap year? True
```

The function `isLeap` can be used to code a function that gives the number of days in a given month:

```
def daysInMonth(month,year):
    if (month==1 or month==3 or month==5 or month==7 or month==8 or␣
 ↪month==10 or month==12):
        return 31
    elif (month==4 or month==6 or month==9 or month==11):
        return 30
    elif (month==2):
        if isLeap(year):
            return 29
        else:
            return 28
    else:
        return 0

print("daysInMonth(2,2020):",daysInMonth(2,2020))
print("daysInMonth(2,2021):",daysInMonth(2,2021))
print("daysInMonth(6,2019):",daysInMonth(6,2019))
print("daysInMonth(3,2020):",daysInMonth(3,2020))
```

```
daysInMonth(2,2020): 29
daysInMonth(2,2021): 28
daysInMonth(6,2019): 30
daysInMonth(3,2020): 31
```

Based on this, the days from January 1st, 1970 can be counted off, until we have fewer days than the number of days in the month. Because it's impossible to know in advance how many months will be traversed, a `while` loop must be used. The condition of this loop is based on what we just designed: this condition is the *negation* of the condition to stop the loop.

```
def actualDay(dfe): # only works with dfe>0
    y = 1970
    m = 1
    d = 1
    mLen = daysInMonth(m,y)
    while dfe>=mLen:      # stop when there are fewer days than the␣
 ↪current month, so continue while there are more
        dfe = dfe-mLen    # remove the days of this month
        m = m+1           # next month
        if m>12:          # maybe a next year
            m = 1
            y = y+1
        mLen=daysInMonth(m,y)
    d = d+dfe             # remaining days after the process
    return (y,m,d)        # return the 3-uple (year,month,day)
```

```
def readableDatetime(zone):
    t = int(time.time()) # t is the number of seconds since epoch (no␣
↪decimal point)
    seconds = t%60
    t = t//60             # t is now the number of minutes since epoch
    minutes = t%60
    t = t//60             # t is now the number of hours since epoch
    t = t+zone            # add the Time Zone difference from GMT
    hours = t%24
    t = t//24             # t is now the number of days since epoch
    date = actualDay(t)
        # With conversion
    #res = str(date[0])+"-"+str(date[1])+"-"+str(date[2])+"␣
↪"+str(hours)+":"+str(minutes)+":"+str(seconds)
        #With formatted string
    res = "%04d-%02d-%02d %02d:%02d:%02d" %␣
↪(date[0],date[1],date[2],hours,minutes,seconds)
    return res

readableDatetime(-4)  # Eastern Summer Time
```

```
'2020-06-09 16:33:28'
```

P5.5.1 Exercise

Part 1: checking date validity

Write a function `validDate` that takes year, month, and day, and returns `True` or `False` based on whether the date is valid. For example:

- `validDate(2020,11,6)` returns `True` because November 6th, 2020 is a valid date
- `validDate(2020,6,31)` returns `False` because June 31st, 2020 is not a valid date
- `validDate(2020,2,29)` returns `True` because February 29th, 2020 is a valid date
- `validDate(2021,2,29)` returns `False` because February 29th, 2021 is not a valid date

```
#
# Code here
#
```

Part 2: getting a valid date from the user

Write a function `inputDate` that asks the user for a date, checks its validity, and asks again until a valid date is entered. It then returns the 3-tuple (year,month,day). For example:

```
Please enter a date:
        Year: 2019
        Month: 4
        Day: 31
```

```
This date does not exists. Please enter a valid date:
        Year: 2021
        Month: 2
        Day: 29
This date does not exists. Please enter a valid date:
        Year: 2020
        Month: 5
        Day: 42
This date does not exists. Please enter a valid date:
        Year: 2020
        Month: 9
        Day: 23
Thank you!
```

```
#
# Code here
#
```

P5.6 Measuring time

To measure time elapsed, the difference between two different time values is used. In the example below, it is used to measure the performance of a program doing some calculations.

```
def performance(n):
    before = time.time()
    power=1
    for i in range(0,n):  # repeat for i in 0, 1, 2, ..., n-1
        power=2*power
    x=0
    pi=3.141592653589793238462643383279502884197169399375105820974
    e=2.718281828459045235360287471352662497757247093699959574966
    for i in range(0,power): # repeat for i in 0, 1, 2, ..., 2^n-1
        x=(x+pi)/e # calculation
    after = time.time()
    return after-before

performance(25) # takes about 2~3 seconds on my computer
```

```
2.3237216472625732
```

P5.6.1 Exercise

Write a function `timedName` that asks the user for his name, and replies with the time it took. Example output (the first `John` being the user's input):

```
What is your name?
John
Hello John, it took you 3.4763026237487793 seconds to answer.
```

```
#
# Code here
#
```

P5.6.2 Exercise

Write a function `sayWhen` that writes out "Tell me when to stop", and continues going until the user actually enters the word *when*. It then outputs the time it took.

For example:

```
Tell me when to stop
        Now
        Stop!
        when
It took you 12.335185289382935 seconds
```

```
#
# Code here
#
```

P5.6.3 Exercise

Write a function `timer` that starts a timer when the user presses the enter key, and records the "lap duration" when the enter key is pressed again, until *Stop* is entered. For example:

```
Press enter to start

Enter "Stop" to stop; press enter to record a lap

Lap 1: 6.43984055519104 seconds

Lap 2: 36.080485105514526 seconds

Lap 3: 58.968928813934326 seconds
Stop
Total:  65.79275012016296 seconds
```

```
#
# Code here
#
```

Project 6
Grade Management with Pandas

Project contents

P6.1 Introduction 129
P6.1.1 Library installation 129
P6.1.2 Importing data 130
P6.2 Column operations 130
P6.2.1 Selecting a column 130
P6.2.2 Functions on a column 131
P6.2.3 Exercise: Range of a column 131
P6.2.4 Creating a derived column from existing columns 132
P6.2.5 Exercise: Regular average 132
P6.2.6 `apply` a function on a column 133
P6.3 Operations on a whole sheet (DataFrame) 134
P6.3.1 Applying a function to all cells 134
P6.3.2 Applying a function to each column (axis=0) 135
P6.3.3 Applying a function to each row (axis=1) 135
P6.4 Filtering 135
P6.4.1 Example : Counting the number of passing students 136
P6.4.2 Exercise: Counting students who got 75 or higher on both their test1 and final 136
P6.5 Saving as an Excel file 136
P6.6 A grade manager 137

P6.1 Introduction

This project uses the `pandas` library which is one of the popular tools for data science. Even though `pandas` has lots of functionalities, this project only focuses on these 3 topics:

- Importing data from an Excel `xlsx` file.
- Updating data
- Saving the updated data into a new Excel file.

P6.1.1 Library installation

To install the required library for this project, execute the following in a terminal:

```
pip3 install -U xlrd
pip3 install -U numpy
pip3 install -U pandas
```

Or in Jupyter:

```
!pip3 install -U xlrd
!pip3 install -U numpy
!pip3 install -U numexpr
!pip3 install -U pandas
```

The libraries are then imported in Python:

```
import numpy as np
import pandas as pd
```

P6.1.2 Importing data

An excel file named `data.xlsx` is provided in the project folder. To import it, use the following command:

`df=pd.read_excel('data.xlsx')`.

Alternatively, it can be downloaded from https://qcckkim.github.io/CS100SAMPLE/data.xlsx:

`df=pd.read_excel('https://qcckkim.github.io/CS100SAMPLE/data.xlsx')`

```
df = pd.read_excel('data.xlsx')
print(df) # Print the spreadsheet
```

```
      name  test1  test2  final
0      bob     70     80     90
1     Jane     75     66     78
2      kim     88     44     77
3    happy     78     65     76
4  holiday     85     76     87
```

P6.2 Column operations

P6.2.1 Selecting a column

A column is accessed by putting the name of the column head in square brackets `[ ]`. For example, to get the `'name'`column:

```
names = df['name']
print(names)
```

```
0        bob
1       Jane
2        kim
3      happy
4    holiday
Name: name, dtype: object
```

P6.2.2 Functions on a column

A column (actually called a *Series* in `pandas` documentation) has many built-in functions. For example, the `.mean()` method computes the average of the column.

```
test1col = df['test1'] # Select a column
avg = test1col.mean()  # Compute the average of that column
print(avg)
```

```
79.2
```

A column behaves like a Python list, hence any function that works on list can be used on a column. For example, the function below computes the average of a list:

```
def avg(L):
  return sum(L)/len(L)

print(avg(test1col)) # Using a list function with a column (Series)
```

```
79.2
```

Other useful functions include finding the maximal value. In this case there is the choice of using the built-in `.max()` method or the standard `max()` function on lists.

For example, to find the maximum score on `test2`:

```
test2col = df['test2'] # Selecting the column (Series object)
m1 = test2col.max()       # .max() method of the Series object
m2 = max(test2col)        # max() function on lists
print(m1,m2)
```

```
80 80
```

For the complete list of built-in functions on columns (`Series`), see the `pandas.Series` documentation.

P6.2.3 Exercise: Range of a column

The range of a list `L` is defined as the difference between `max(L)` and `min(L)`:

```
max(L)-min(L)
```

Similarly, the range of a column `C` is `max(C)-min(C)`.

In the Excel files there are three columns corresponding to tests; the goal is to determine the greatest range of the three. This can be broken down as follows:

1. Define the `therange(L)` function that takes a list `L` and returns its range.
2. Compute the three ranges of columns `test1`, `test2`, and `final`, using `therange(L)` function .
3. Store these three ranges in a list named `ranges`.
4. Find the maximum of the ranges stored in `ranges`.

```
#
# Code here
#
```

P6.2.4 Creating a derived column from existing columns

A new column can be created from existing columns.

Assuming the grading rule is as follows:

- test1: 25%
- test2: 35%
- final: 40%

Then, for example, bob's class average is $70 * 0.25 + 80 * 0.35 + 90 * 0.40 = 81.5$

This average can be calculated for all students and stored in a new `avg` column.

```
df['avg'] = df['test1']*0.25+df['test2']*0.35+df['final']*0.40 # Defining␣
↪a new column
print(df['avg']) # Print the new colum
df               # Return the whole sheet
```

```
0    81.50
1    73.05
2    68.20
3    72.65
4    82.65
Name: avg, dtype: float64
```

	name	test1	test2	final	avg
0	bob	70	80	90	81.50
1	Jane	75	66	78	73.05
2	kim	88	44	77	68.20
3	happy	78	65	76	72.65
4	holiday	85	76	87	82.65

P6.2.5 Exercise: Regular average

A teacher wants to compute a *regular average* using the following rule for each student.

- test1: 42%
- test2: 58%

Make a new `regularavg` column which contains the regular average of each student.

```
#
# Code here
#
```

P6.2.6 `apply` a function on a column

The `.apply()` method of columns takes a function as an argument and applies it to all the elements of the column. The *result is a column*, and can therefore be stored in the data as a new column.

For example here the function `f` is applied to `column1` to create `column2`:

```
df['column2']=df['column1'].apply(f)
```

Example: `pass` column from `avg` column

The average column can be used to decide whether a student passed or failed, using the following pass/fail rule:

- If the average is greater or equal to 70, return `True`.
- Otherwise, return `False`.

First, the `passfail(avg)` function must be defined:

```
def passfail(avg):
  if avg >= 70:
      return True
  else:
      return False
```

Then the `pass` column is created by using the `.apply()` method of the existing `avg` column:

```
df['pass'] = df['avg'].apply(passfail) # Function passfail is given as␣
 ↪argument
print(df['pass'])
df
```

```
0     True
1     True
2    False
3     True
4     True
Name: pass, dtype: bool
```

	name	test1	test2	final	avg	regularavg	pass
0	bob	70	80	90	81.50	75.80	True
1	Jane	75	66	78	73.05	69.78	True
2	kim	88	44	77	68.20	62.48	False
3	happy	78	65	76	72.65	70.46	True
4	holiday	85	76	87	82.65	79.78	True

The table shows that kim failed the class.

Exercise: Letter grade

Create a new `grade` column from the `avg` column using the following grade rule:

- If $90 \leq avg \leq 100$: A
- If $75 \leq avg < 90$: B
- If $60 \leq avg < 75$: C
- Otherwise ($avg < 60$): F

```
#
# Code here
#
```

P6.3 Operations on a whole sheet (DataFrame)

A `DataFrame` is the name given by `pandas` to the whole spreadsheet. The `.apply()` method also exists for `DataFrames`, and can be used in three differently ways.

First, let's create a new DataFrame `df2` from a list `L`:

```
L=[  # L is a list of lists
   [1,2], # Each list is a row
   [3,4],
   [5,6]
]
# Create a sheet (DataFrame) with column names 'x' and 'y'
df2 = pd.DataFrame(L,columns=['x','y'])
df2
```

```
   x  y
0  1  2
1  3  4
2  5  6
```

P6.3.1 Applying a function to all cells

The `.apply()` method using only a function as argument executes the function on each cell and returns a new sheet. In the example below the `np.square` function (that calculates the square of a number) is applied to `df2`.

```
df2squared = df2.apply(np.square)
print(df2,df2squared,sep="\n")
```

```
   x  y
0  1  2
1  3  4
2  5  6
    x   y
0   1   4
1   9  16
2  25  36
```

Any function that is defined can be applied this way:

```
def sq(x):
  return x**2
df2.apply(sq)
```

```
    x   y
0   1   4
1   9  16
2  25  36
```

P6.3.2 Applying a function to each column (axis=0)

By setting the keyword argument `axis` to `0`, the apply operation will be performed on columns: the function given as a parameter must operate on columns (or lists). The values returned by the successive application of the function on columns are returned into a row.

For example, applying the `np.sum` function to each column of `df2` returns the row with a value for each column. As the name indicates, the `np.sum` adds all the values in the list, so the results are $1 + 3 + 5 = 9$ for `x` and $2 + 4 + 6 = 12$ for `y`.

```
df2.apply(np.sum,axis=0)
```

```
x     9
y    12
dtype: int64
```

P6.3.3 Applying a function to each row (axis=1)

This is similar to the above, but using a function operating on rows to create a column. In that case the `axis` argument must be set to `1`.

Below `np.sum` function is applied to each row of `df2` and the column of results is returned.

```
df2.apply(np.sum,axis=1)
```

```
0     3
1     7
2    11
dtype: int64
```

P6.4 Filtering

Filtering is the selection of some rows from a `DataFrame` based on the values it contains. Filtering is used to collect a meaningful sample from existing data.

The syntax is as follows:

```
df[ column condition  ]
```

P6.4.1 Example : Counting the number of passing students

We filter the class by only selecting students who passed. The condition, here pertaining to the `'pass'` column, is contained within `df[    ]`.

```
dfpass = df[df['pass']==True]
dfpass
```

```
      name  test1  test2  final    avg  regularavg  pass grade
0      bob     70     80     90  81.50       75.80  True     B
1     Jane     75     66     78  73.05       69.78  True     C
3    happy     78     65     76  72.65       70.46  True     C
4  holiday     85     76     87  82.65       79.78  True     B
```

The `.count()` method on `Series` returns the number of rows for this column. Applying it to the filtered `DataFrame` allows to compute the number of passing students.

```
dfpass['pass'].count()
```

```
4
```

P6.4.2 Exercise: Counting students who got 75 or higher on both their test1 and final

Count students who get 75 or higher on their test1 and final.

1. Filter `df` using the above conditions.
2. Choose any column from the filtered `df` and count using `count()`.

Note: If there are two conditions to filter, they can be combined using `&` for *and*, `|` for *or*.

```
df[ column cond1 & column cond2 ] # cond1 and cond2
df[ column cond1 | column cond2 ] # cond1 or cond2
```

```
#
# Code here
#
```

P6.5 Saving as an Excel file

A `DataFrame` can be saved as a sheet of an Excel document. In the example below, the resulting sheet is saved into Sheet1 of file `result.xlsx`.

```
writer = pd.ExcelWriter('result.xlsx') # Open the file for writing
df.to_excel(writer,'Sheet1')           # Write df into Sheet1
writer.save()                          # Save
```

P6.6 A grade manager

1. Import the data from file `PROJECT.xlsx`, provided in the project folder or online at https://qcckkim.github.io/CS100SAMPLE/PROJECT.xlsx.

2. Create `'avg'` column which contains the class average using the following grading rule.

   - hw : 10%
   - test1: 15%
   - test2: 15%
   - test3: 15%
   - test4: 15%
   - final: 30%

3. Create 'grade' column with letter grade using `avg` column from the following grading rule.

   - `avg` ≥ 90 : A
   - `avg` ≥ 80 : B
   - `avg` ≥ 70 : C
   - `avg` ≥ 60 : D
   - `avg` < 60 : F

4. Count the number of A grade, B grade, C grade, D grade and F grade students using the `count` function.

5. Count the number of students who obtained both an average > 70 and > 80 for the final.

```
#
# Code here
#
```

Project 7
Descriptive Statistics and Histogram of Frequencies

P7.1 Introduction

The main objective of this project is to explain concepts about descriptive statistics and teach how to calculate measures of *central tendency* and *variability/dispersion* using Python.

P7.1.1 Central Tendency

Central Tendency is a way of producing a single value which represents a *central* value in a set of values.

Mean, Median and Mode are the most commonly used three measures of central tendency. They are different kinds of *averages*.

Mean: The *mean* is a measure which is most commonly used to describe the *average* value and calculated by dividing the sum of values by the number of values in a list of data values.

$$Mean = \frac{\sum x}{N}$$

Here, $\sum$ represents summation, N represents number of values and x represents the values.

Example:

Using the `sum()` and `len()` functions from standard library to calculate the mean:

```
Grades = [34, 65, 23, 89, 59]
Mean = sum(Grades)/len(Grades)
print("Mean =", Mean)
```

```
Mean = 54.0
```

Median: The *median* is the middle value in a sorted list of data values. If the number of values in the list is even, then the average of the two middle values is used as the median.

If the total number of values is odd:

$$Median = \left(\frac{n+1}{2}\right)^{th} term$$

If the total number of values is even:

$$Median = \frac{\left(\frac{n}{2}\right)^{th} term + \left(\frac{n}{2}+1\right)^{th} term}{2}$$

Example:

Using the `sorted()` and `len()` functions from standard library to calculate the median:

```
grades = [43, 65, 26, 45, 12, 67]
sortedgrades = sorted(grades)
print("Sorted grades =", sortedgrades)
if(len(sortedgrades) % 2):
    median = sortedgrades[len(sortedgrades)//2]
else:
    median = (sortedgrades[len(sortedgrades)//2] +
 ↪sortedgrades[len(sortedgrades)//2 -1])/2
print("Median =", median)
```

```
Sorted grades = [12, 26, 43, 45, 65, 67]
Median = 44.0
```

Mode: The *mode* is the most frequently occurring value in a list of data values. There can be multiple modes or no mode in a list of values.

Python's statistics module has functions that calculates mean, median and mode.

Example:

Using `statistics` module to calculate mean, median and mode:

```
import statistics
grades = [1, 3, 5, 7, 2, 4, 7, 2, 9, 3, 7, 6, 5]
mymean = statistics.mean(grades)
mymedian = statistics.median(grades)
mymode = statistics.mode(grades)
print("Mean =", mymean,"\t", "Median =", mymedian, "\t","Mode =", mymode)
```

```
Mean = 4.6923076923076925        Median = 5      Mode = 7
```

P7.1.2 Dispersion/Variability

Dispersion or *Variability* are measures that help understand how spread out the values are in a data set.

Variance and *Standard deviation* are two measures of *dispersion*.

The **Variance** is sometimes described as "the mean of distance to the mean". It is calculated using the following steps:

- Calculate the mean of the dataset
- Subtract the mean from every data value
- Get the squares of the difference
- Calculate the mean of the squares

Formula:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}$$

In the above equation, σ^2 is variance, x_i is ith value, N is the number of values and μ is the mean value of the dataset.

Standard Deviation is the square root of variance.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}$$

The smaller the variance and standard deviation are, the closer the data values are to the mean and the less overall dispersion.

Example:

Calculating variance and standard deviation with `statistics` module:

```
from statistics import pvariance, pstdev
grades = [1, 3, 5, 7, 2, 4, 7, 2, 9, 3, 7, 6, 5]
myvariance = pvariance(grades)
mystdev = pstdev(grades)
print('Variance = {:2.3}    StDev = {:2.3}'.format(myvariance, mystdev))
```

```
Variance = 5.44    StDev = 2.33
```

P7.2 Creating Bar Graphs with `matplotlib` and `seaborn` modules

A single column bar graph consists of the following elements, as depicted in Figure P7.1.

In Python the `matplotlib` and `seaborn` modules can create such graphs. First, these libraries may not be installed by default so, the following command can be run in a terminal to install the modules:

```
pip3 install -U matplotlib
pip3 install -U seaborn
```

Or use the following in Jupyter:

```
!pip3 install -U matplotlib
!pip3 install -U seaborn
```

```
# Include the necessary modules
import statistics as st
import seaborn as sns
import matplotlib.pyplot as plt

# First create a list of values
votes = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 5]
```

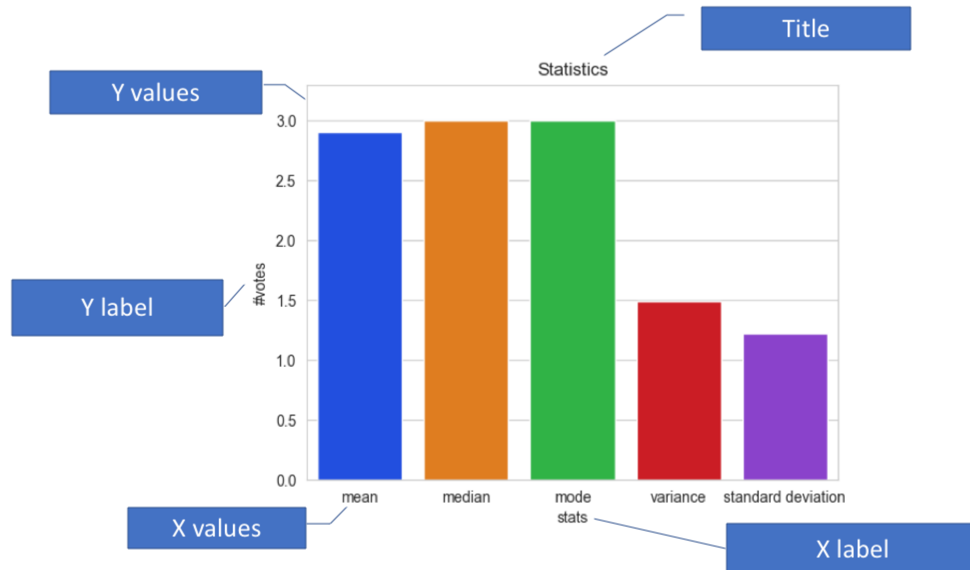
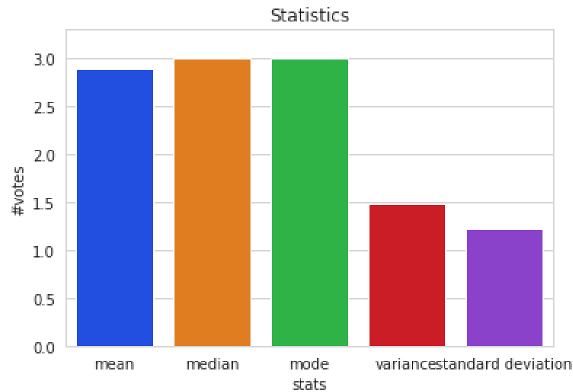


Figure P7.1: The elements of a bar graph.

```
# Using the statistics module, calculate mean, median, mode, variance and␣
↪standard deviation
mean = st.mean(votes)
median = st.median(votes)
mode = st.mode(votes)
var = st.pvariance(votes)
std = st.pstdev(votes)

# Create the lists for x values and y values
yvalues = [mean, median, mode, var, std]
xvalues = ['mean', 'median', 'mode', 'variance', 'standard deviation']

# Set the graph properties
title = 'Statistics'
sns.set_style('whitegrid')

# Set the x and y values
axes = sns.barplot(x = xvalues, y=yvalues, palette= 'bright')
axes.set_title(title)                           # Set the title of the graph
axes.set(xlabel= 'stats', ylabel= '#votes')  # Set the x and y labels
axes.set_ylim(top=max(yvalues)*1.10)          # Set the maximum y value in␣
↪the graph
plt.show()                                      # Display the graph
```



P7.3 Exercise: Creating the Histogram of a Dataset

P7.3.1 Importing the modules

In this exercise, `random`, `statistics`, `matplotlib.pyplot` and `seaborn` libraries are to be used to create a histogram graph in Python.

Include the necessary modules.

```
#
# Code here
#
```

What is a histogram? A *histogram* is a frequency graph. The values in the x axis represent the unique values in a dataset. Each corresponding y value represents how many times this particular data value occurs in the dataset.

For example with the dataset: `[1, 3, 5, 6, 2, 3, 4, 6, 2, 2, 1, 3]`

The X values is the set of unique values:`[1, 2, 3, 4, 5, 6]`

The Y values are the frequency of each value: `[2, 3, 3, 1, 1, 2]` (1 appears 2 times, 2 appears 3 times, ...)

The histogram for the above set of values looks like the one on Figure P7.2.

P7.3.2 Creating a random dataset

Create a dataset of 50 integer values. The dataset is to be stored in a list named `votes`.

You may create them randomly within a range of `[1:10]` using the `randint(left, right)` function from the `random` module. Note that in computers, "random numbers" are not really random, but generated from a seed. Changing the seed value below will change the generated random values.

```
rd.seed(0) # Change the value here
#
# Code here
#
```

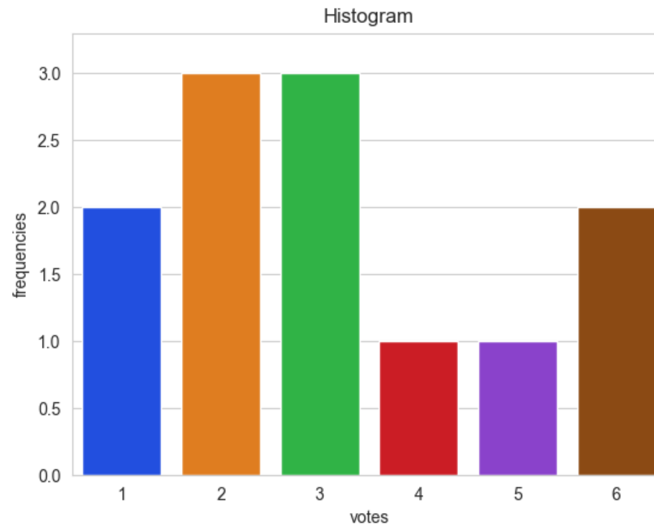


Figure P7.2: A histogram representing frequencies of the dataset `[1,3,5,6,2,3,4,6,2,2,1,3]`.

P7.3.3 Calculating central tendency and dispersion

1. Calculate the mean, median, mode, variance and standard deviation using the `statistics` module.
2. Create a bar graph to display the results. Use `matplotlib.pyplot` and `seaborn` modules.

```
#
# Code here
#
```

P7.3.4 Removing outliers

An *outlier* is a data value that is out of the ordinary and possibly way out of the ordinary. They are sometimes called noise or bad data and can distort the results. Usually, values that are *3 standard deviations* away from the mean of the data are considered outliers.

For example, if the mean of the dataset is 20 and standard deviation is 4, values that are below $20 - 3 \times 4 = 8$ or above $20 + 3 \times 4 = 32$ are considered outliers. In this case the value of 1 could be an outlier since it is less than 8.

1. Define a function that finds the outliers.
2. Print the outliers in the `votes` data set.

```
#
# Code here
#
```

P7.3.5 Calculating the frequencies

Calculate the frequencies of unique values in your dataset. Create a bar histogram to display your results. In order to do that:

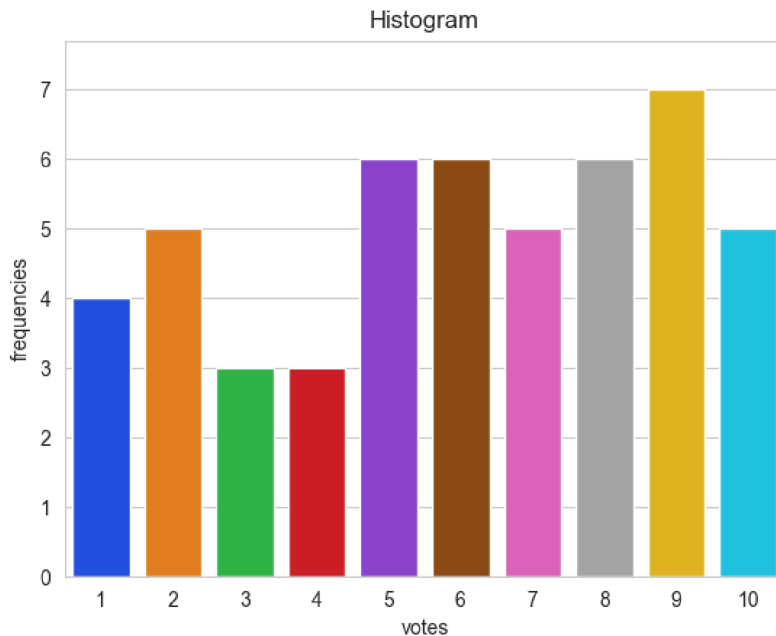


Figure P7.3: A histogram representing frequencies of a random sample.

1. Write a function `unique()` that that takes a list as a parameter and returns unique values in the list as a new list. Use the `in` keyword to search and remove duplicates and store them in a new list.
2. Write a second function `frequencies()` that takes this unique list and your original dataset as parameters and returns the frequency list as a result.
3. Use the returned list values to create your bar graph as x values and y values.

The resulting bar graph may look like the one on Figure P7.3.

```
#
# Code here
#
```

Project 8
Emotion Analysis

Project contents

P8.1 Introduction **147**
P8.1.1 Installing the libraries 147
P8.2 Cleaning Text Data **148**
P8.2.1 Reading a file into a string variable 148
P8.2.2 Tokenizing a string into words 149
P8.2.3 Identifying punctuation 149
P8.2.4 Identifying stopwords 150
P8.2.5 Converting to lowercase 150
P8.2.6 Stemming the words 150
P8.3 Measuring Polarity and Subjectivity **151**
P8.4 Exercise **151**
P8.4.1 Step 1: Cleaning the data 152
P8.4.2 Step 2: Finding the word count of the books 153
P8.4.3 Step 3: Finding the number of occurrences of the emotion words in the books 153
P8.4.4 Step 4: Creating a graph that compares the number of occurrences of emotion words 155
P8.4.5 Step 5: Creating a bar graph comparing the polarity and subjectivity 156

P8.1 Introduction

The objective of this project is to analyze text from Project Gutenberg Library and compare them. Do they mostly convey positive or negative emotions? Are they subjective? How about the feelings of anger, happiness, satisfaction, confusion, urgency and helplessness? How do they compare to each other based on these emotions? The answers to these questions will be given using the natural language processing libraries of Python.

P8.1.1 Installing the libraries

In this project, Python's `nltk`, `textblob`, and `pandas` modules will be used. The `nltk` module has several functions which can be used for cleaning the data. The `textblob` module provides some text analysis tools. The `pandas` and `matplotlib` modules are used to draw charts.

They might not be installed by default so, the following command can be run in a terminal to install the modules:

```
pip3 install -U nltk
pip3 install -U textblob
pip3 install -U pandas
pip3 install -U matplotlib
```

Or use the following in Jupyter:

```
!pip3 install -U nltk
!pip3 install -U textblob
!pip3 install -U pandas
!pip3 install -U matplotlib
```

After that, the `stopwords` and `punkt` packages need to be downloaded using the following script:

```
import nltk
import ssl

try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context

nltk.download("stopwords")
nltk.download("punkt")
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/esmayildirim/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/esmayildirim/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
True
```

P8.2 Cleaning Text Data

P8.2.1 Reading a file into a string variable

`open()` function can be used to create a handle to a file before reading it. It requires two parameters: 1) file path 2) read/write purposes.

The following example reads a file in the working directory where the Python scripts is running. Assume that `myfile.txt` already exists in this path.

```
file_handle = open('myfile.txt','rt')
```

In the above snippet, `rt` stands for opening the file `myfile.txt` to `read text`. Only filename is sent as an argument rather than a full path such as `/home/ey/projects/myfile.txt`. The path naming convention changes from Unix/Linux to Windows systems. In Windows, the path can be something like: `C:\Users\ey\projects\myfile.txt`. To give a correct path, one should know where the file is located in the system.

After opening the file, it can be read with the `read()` method of the file handle. In the following snippet, `myvar` string variable contains the entire contents of the file:

```
myvar = file_handle.read()
# Don't forget to close the file once it has been read
file_handle.close()
print(myvar)
```

```
Here are the contents of my shopping list: milk, eggs, flour, apples,
↪oranges.
```

P8.2.2 Tokenizing a string into words

To split the string into words, `word_tokenize()` function from the `nltk` module is used. The function accepts a string variable as a parameter and returns a list of words.

For a refresher on lists see Lecture L7.1.

```
from nltk import word_tokenize
tokens = word_tokenize(myvar)
print(tokens)
```

```
['Here', 'are', 'the', 'contents', 'of', 'my', 'shopping', 'list', ':',
↪'milk', ',', 'eggs', ',', 'flour', ',', 'apples', ',', 'oranges', '.']
```

P8.2.3 Identifying punctuation

In the previous step, the tokenizer considers punctuation marks as words. But they should be removed. A string variable has a method called `isalpha()` to check whether the characters in the string are alphabets. If so, the method returns True, otherwise it returns False. The returned value can be used to remove these tokens from the list.

```
for token in tokens:
    print(token, ":", token.isalpha())
```

```
Here : True
are : True
the : True
contents : True
of : True
my : True
shopping : True
list : True
: : False
milk : True
```

```
, : False
eggs : True
, : False
flour : True
, : False
apples : True
, : False
oranges : True
. : False
```

P8.2.4 Identifying stopwords

Stopwords are commonly used words in a language such as subjects like `I`, `you`, `we` or question words like `when`, `how`, or auxilary verbs like `am`, `is` , `are` and so on... These words are generally removed before any text processing algorithm is applied.

`nltk` module has a package called `corpus` that contains stopwords in the English language. This list can be compared to the words in the list obtained from the tokenized text:

```
from nltk.corpus import stopwords
my_stop_words = stopwords.words('english')
for token in tokens:
    if token in my_stop_words:
        print(token, 'is a stopword.')
```

```
are is a stopword.
the is a stopword.
of is a stopword.
my is a stopword.
```

P8.2.5 Converting to lowercase

To make sure that same word in uppercase letters and lowercase letters are treated as the same, it is common to convert all characters in a word to lowercase. The `lower()` method of a string variable returns the lowercase version of the string.

```
word = 'HelLo'
print(word.lower())
```

```
hello
```

P8.2.6 Stemming the words

Two words with different structures may come from the same stem. For example, the words `identifying` and `identified` come from the same stem: `identifi`. Stemming makes sure that these two words are treated as the same word. The `nltk.stem.porter` package has a class named `PortStemmer` that has a method named `stem()` to be used to stem words. First, an object of the class need to be created. Then the method can be called on the object.

```
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer()     # object is created
word_list = ['identifying', 'identified']
for word in word_list:
    print(porter.stem(word)) #stem() method is called
```

```
identifi
identifi
```

P8.3 Measuring Polarity and Subjectivity

Python's `textblob` module has objects that can measure `polarity` and `subjectivity` in a text. `Polarity` is a number between -1 and 1. If the measure gives a positive number then the sentiment of the text is mostly positive. Otherwise, it is negative. A value of 0 shows neutral sentiment. Subjectivity is a number between 0 and 1. 0 refers to being objective while 1 refers to being subjective.

To use the methods, a `TextBlob` object has to be created. Its data members `sentiment.polarity` and `sentiment.subjectivity` provide these measures.

```
from textblob import TextBlob
text = 'I am so happy that you found your favorite book.'    # A positive␣
↪and subjective sentence
blob_object = TextBlob(text)
print("Polarity:", blob_object.sentiment.polarity)
print("Subjectivity:", blob_object.sentiment.subjectivity)
text = 'People hate tardiness. So, stop behaving like that.' # A negative␣
↪and less subjective sentence
blob_object = TextBlob(text)
print("Polarity:", blob_object.sentiment.polarity)
print("Subjectivity:", blob_object.sentiment.subjectivity)
```

```
Polarity: 0.65
Subjectivity: 1.0
Polarity: -0.8
Subjectivity: 0.9
```

P8.4 Exercise

Before starting to analyze your favorite books, the text version of the books must be downloaded from Project Gutenberg site. Go to: https://www.gutenberg.org and download `Plain Text UTF-8` version of the two/or more books from two of your favorite authors.

P8.4.1 Step 1: Cleaning the data

Step 1.1: Defining the function

Write a function `clean_text()` that will accept a `filepath` string variable as a parameter and return a list of words that will have gone through tokenizing, removal of punctuation, removal of stopwords, conversion to lowercase and stemming processes.

```
def clean_text(filepath):
    # Open file by using its path and read the contents into a string␣
↪variable
    #
    # Complete the code here
    #
    # Convert the string variable into tokens
    #
    # Complete the code here
    #
    # Remove punctuation from the tokens
    #
    # Complete the code here
    #
    # Convert words to lowercase
    #
    # Complete the code here
    #
    # Remove stopwords
    #
    # Complete the code here
    #
    # Stem the words
    #
    # Complete the code here
    #
    # Return the words as a list
    return words
```

Step 1.2: Using the function on the book

Use the above function to clean text data for the books: Call the `clean_text` function once for each book text data file, assigning the returned lists into variables of your choosing.

```
#
# Code here
#
```

Step 1.3: Using the function on the emotion words

Inside the project folder, there is another folder named `emotion_word_data` which contains text files which also contain words for different emotions. Table P8.1 shows excerpts from

the contents of these text files.

Use `clean_text()` function to clean them as well. Assign the returned lists into variables that represents the emotions in the title of the text files.

P8.4.2 Step 2: Finding the word count of the books

1. Code the function `find_word_count()` that computes the word counts of the books by using a dictionary. The `key->value` pair of each dictionary must refer to `word->number of occurrences` in the books. The cleaned word lists may be used to do that. The `find_word_count()` function accepts a word list as an argument and returns a dictionary of word count.

For a refresher on dictionaries see Lecture L7.3.

```
def find_word_count(words):
    # words is the list of cleaned words
    #
    # Code the function here
    #
    return dct_book
```

2. Call the function once for all the books. Assign the returned dictionaries proper names.

P8.4.3 Step 3: Finding the number of occurrences of the emotion words in the books

Write a function to create the *emotion index* which is basically the total number of occurrences of words of each emotion lists in the books. The function takes two parameters: `index_name` is the name of the emotion (e.g. "Anger"), `index_words` is a list which contains the cleaned emotion word list (e.g. list for the anger.txt words), `dictionary` is the word count dictionary of a book created in the previous step.

```
def emotion_index(index_name, index_words, dictionary):
    # index_words is a list which contains the cleaned emotion words,
    # dictionary contains the word-> number of occurrences pairs of the␣
↪book
    #
    # Code the function here
    #
    return total # The total number of occurrences of the emotion words␣
↪in the book words
```

Use `emotion_index()` function to find the number of occurrences for each emotion by using the cleaned emotion words lists for the two books you would like to compare. You should be creating 14 emotion indices (7 emotions per 2 books). Append the returned numbers to a list for each book. Example names for the lists could be `pap_list` and `atotc_list`.

```
# Add the results to a list to be used in graphing
print("Emotion Indices for 'Pride and Prejudice'")
#
```

anger.txt	confusion_helplessness.txt	happy_alive.txt	inspired.txt
Ordeal	doubtful	blissful	motivated
Outrageousness	uncertain	joyous	eager
Provoke	indecisive	delighted	keen
Repulsive	perplexed	overjoyed	earnest
Scandal	embarrassed	gleeful	inspired
Severe	hesitant	thankful	enthusiastic
Shameful	disillusioned	festive	bold
Shocking	distrustful	ecstatic	brave
Terrible	misgiving	satisfied	daring
Tragic	unsure	cheerful	hopeful
...	...	...	...

relaxed_peaceful.txt	safe_satisfied.txt	urgency.txt
calm	Accurate	Magical
at ease	instantly	Instantly
comfortable	Advantage	Magnificent
content	Always	Miracle
quiet	Bargain	Important
certain	Certain	Profitable
relaxed	Certainly	Proven
serene	Confident	Quick
bright	Convenient	Remarkable
blessed	Definitely	Results
...	...	...

Table P8.1: Excerpts from the emotion words files.

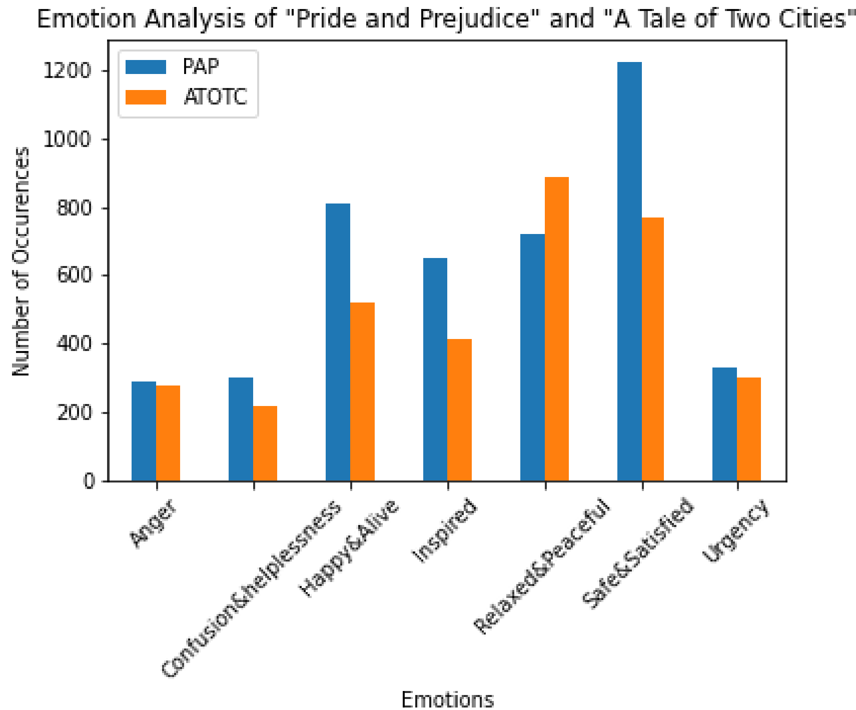


Figure P8.1: Graph bar showing emotion analysis of *Pride and Prejudice* (PAP) and *A Tale of Two Cities* (ATOTC).

```
# Code here
#
# Repeat similar operations for the other novel
print("\nEmotion Indices for 'A Tale of Two Cities'")
#
# Code here
#
```

P8.4.4 Step 4: Creating a graph that compares the number of occurrences of emotion words

Create a multi-column bar graph to compare the two books by the emotions using `pandas` and `matplotlib.pyplot` modules of Python. An example graph is given on Figure P8.1.

In this graph, the books *Pride and Prejudice* by Jane Austen and *A Tale of Two Cities* by Charles Dickens are compared. The X-axis of the graph shows the emotions while the Y-Axis shows the number of occurrences of emotion words in the books.

Complete the code below to create a Pandas dataframe and plot the graph of emotion analysis.

The `pandas` and `matplotlib` libraries were used in Project P7.

```
import pandas as pd
import matplotlib.pyplot as plt

# Here emotions is a list that contains the emotion names such as anger
```

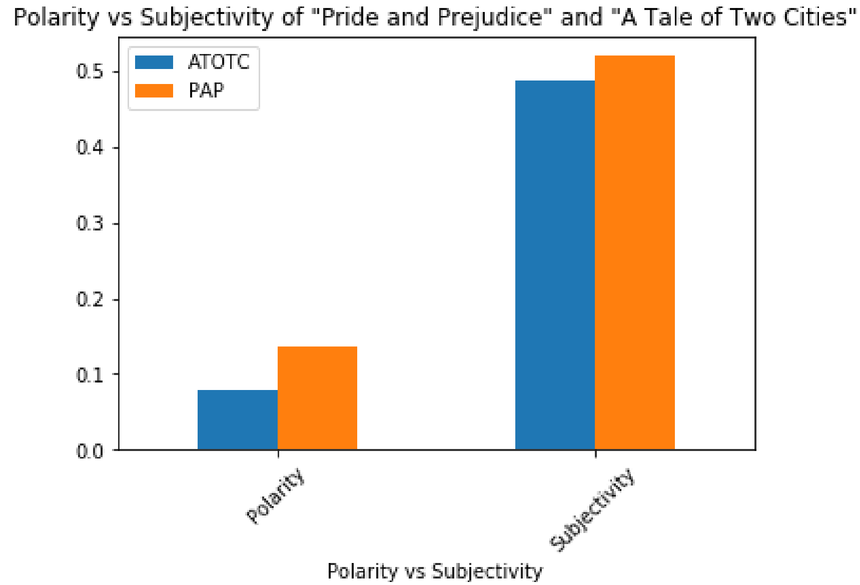


Figure P8.2: Polarity and subjectivity analysis of *Pride and Prejudice* (PAP) and *A Tale of Two Cities* (ATOTC).

```
# You may use any name you wish for your lists
emotions = ['Anger','Confusion&helplessness', 'Happy&Alive',␣
↪'Inspired','Relaxed&Peaceful', 'Safe&Satisfied','Urgency']
plotdata = pd.DataFrame({"PAP":pap_list, "ATOTC":atotc_list}, index =␣
↪emotions)
#
# Code here
#
```

Write your observations about the books and authors. For example, based on the graph in Figure P8.1, `Pride and Prejudice(PAP)` contains more words of Anger, Confusion&Helplessness, Happy, Inspired, Safe&Satisfied and Urgency while `A Tale of Two Cities(ATOTC)` contains more words for relaxed and peaceful. Of course, these results could be normalized by dividing each count by the total number of words in each book. Try this approach and see if there is a change in the results. Both authors are British and the books are from similar eras(published in 1813 and 1859 respectively). While PAP is about relationships between men and women, ATOTC is more about war years. One would expect ATOTC to include more words of `anger` and less words of `relaxed&peaceful`. Other than that it looks like PAP has more positivity in it then ATOTC. Write your own conclusions about your books.

P8.4.5 Step 5: Creating a bar graph comparing the polarity and subjectivity

Use the `textblob` module to find the polarity and subjectivity of your books and create a bar graph like the one depicted on Figure P8.2.

Write your observations. For example, in the graph of Figure P8.2, PAP is listed as

a more positive and subjective book compared to ATOTC which also proves our previous findings from the emotion words. But polarity is close to 0, so they are close to neutral as well. How about your books? Are they positive, negative, subjective or objective?

```
#
# Code here
#
```

Project 9
Dynamics on Functions

Project contents

P9.1 The mathematical setting **159**
P9.1.1 Self composition 159
P9.1.2 The n-th iterated function 160
P9.2 Orbit of x **160**
P9.2.1 Example: n-th Orbit of 1 for $f(x) = 2x + 1$ 160
P9.2.2 Exercise 162
P9.3 The Collatz Conjecture **162**
P9.3.1 Total Stopping Time 163
P9.3.2 Exercise: Defining $h(x)$ 163
P9.3.3 Exercise: Defining `totalstoppingtime(x)` 163
P9.3.4 A scatter plot from a list of points 164
P9.3.5 Exercise: A scatter plot for the Total Stopping Time 164

P9.1 The mathematical setting

Let f be a mathematical function. Even though f can accept any numbers and returns any numbers, it is assumed that f is defined on the set of positive integer, $\mathbb{Z}_{>0}$ and returns positive integers.

Therefore f is a function from $\mathbb{Z}_{>0}$ to $\mathbb{Z}_{>0}$:

$$f : \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$$

P9.1.1 Self composition

f can be composed with f itself. Namely, the 2-th iterated function of f is defined as follow:

$$f^2(x) = f(f(x))$$

Be aware that $f^2(x)$ is **not** $f(x) \cdot f(x)$.

For example, if $f(x) = 2x + 1$, then

$$f^2(x) = f(f(x)) = 2(f(x)) + 1 = 2(x + 1) + 1 = 4x + 2 + 1 = 4x + 3$$

But

$$f(x) \cdot f(x) = (2x + 1)(2x + 1) = 4x^2 + 4x + 1$$

P9.1.2 The n-th iterated function

In general, we define n-th iterated function, f^n recursively.

$$f^n(x) = \begin{cases} id, & \text{for } n = 0 \\ f^{n-1}(f(x)), & \text{for } n > 0 \end{cases}$$

where id is the *identity function*: $id(x) = x$.

For example, for $f(x) = 2x + 1$,

$$f^3(x) = f^2(f(x)) = 4(f(x)) + 3 = 4(2x + 1) + 3 = 4x + 7$$

Exercise

Let g be the function from $\mathbb{Z}_{>0}$ to $\mathbb{Z}_{>0}$ and defined by

$$g(x) = 2x$$

Compute $g^2(x)$ and $g^3(x)$.

P9.2 Orbit of x

For $x \in \mathbb{Z}_{>0}$, the sequence of values $f^n(x)$ is called the orbit of x.

$$\{x, f(x), f^2(x), f^3(x), \ldots\}$$

We define $x_i = f^i(x)$, so: $x_0 = x$, $x_1 = f^1(x)$, $x_2 = f^2(x)$, ..., $x_n = f^n(x)$. Then the orbit of x is given by the sequence $\{x_i\}_{i \geq 0}$,

$$x_0 = x, x_1 = f(x_0), x_2 = f(x_1), x_3 = f(x_2), \ldots$$

We also define the n-th orbit of x for f as the finite sequence

$$x_0, x_1, \ldots, x_n$$

P9.2.1 Example: n-th Orbit of 1 for $f(x) = 2x + 1$

The orbit of 1 for $f(x) = 2x + 1$ is

$$\{1, f(1) = 2(1) + 1 = 3, f(2) = 2(3) + 1 = 7, f(3) = 2(7) + 1 = 15 \ldots\} = \{1, 3, 7, 15, \ldots\}$$

The 3rd orbit of 1 for $f(x) = 2x + 1$ can be computed using Python.

1. Define a Python function for `f`.

```
def f(x):
    return 2*x+1
```

2. Compute $x_0, x_1, \ldots, x_n$ when $x_0 = 1$ and $n = 3$.

```
x0 = 1
x1 = f(x0)
x2 = f(x1)
x3 = f(x2)
print(x0,x1,x2,x3)
```

```
1 3 7 15
```

3. Gather the x_i into a Python list `X`.

```
X = [1]            # Initialize the list with element 1
X.append(f(X[0])) # Add x1=f(1)=f(x0) to the end of the list
X.append(f(X[1])) # Add x2=f(f(1))=f(x1) to the end of the list
X.append(f(X[2])) # Add x3=f(f(f(1)))=f(x2) to the end of the list
```

4. Make the code cleaner.

- Indexes of X can be replaced with a variable `i`.

```
X = [1]
i = 0
X.append(f(X[i]))
i = 1
X.append(f(X[i]))
i = 2
X.append(f(X[i]))
```

- The line `X.append(f(X[i]))` is repeated several times, only the value of `i` changes. This is better done using a loop.

```
# Version using a while loop
X = [1]
i = 0
while i<3:
    X.append(f(X[i]))
    i = i+1
print(X)
```

```
[1, 3, 7, 15]
```

```
# Version using a for loop
X = [1]
for i in range(0,3):
    X.append(f(X[i]))
print(X)
```

```
[1, 3, 7, 15]
```

P9.2.2 Exercise

Write a python function `orbit(f,x,n)` which returns the n-th orbit of `x` for `f`. Note that a function can be given as an argument of a function.

For example: computing the 7-th orbit of 3 for $f(x) = 3x + 2$.

```
def f(x):          # define f
  return 3*x+2
R = orbit(f,3,7) # use f as the argument of the orbit function
print(R)
```

The output is

```
[3, 11, 35, 107, 323, 971, 2915, 8747]
```

```
#
# Define the orbit function here
#

def f(x):
    return 3*x+2
R = orbit(f,3,7)
print(R)
```

P9.3 The Collatz Conjecture

Let h be the function from $\mathbb{Z}_{>0}$ to $\mathbb{Z}_{>0}$ as follows:

$$h(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ 3x+1, & \text{if } x \text{ is odd} \end{cases}$$

Consider the corresponding orbit of x for h,

$$x_0 = x,\ x_1 = h(x_0),\ x_2 = h(x_1),\ \ldots$$

For example, the orbit of 3 for h can be computed as follows:

$$x_0 = 3 \qquad x_1 = 3(3)+1 = 10 \qquad x_2 = \frac{10}{2} = 5 \qquad x_3 = 3(5)+1 = 16$$

$$x_4 = \frac{16}{2} = 8 \qquad x_5 = \frac{8}{2} = 4 \qquad x_6 = \frac{4}{2} = 2 \qquad x_7 = \frac{2}{2} = 1$$

$$x_8 = 3(1)+1 = 4 \qquad x_9 = \frac{4}{2} = 2 \qquad x_{10} = \frac{2}{2} = 1 \qquad \ldots$$

The orbit of 3 for h is therefore

$$3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$$

Once $x_i = 1$ for some $i \geq 0$, the orbit repeats with the pattern $4, 2, 1$. The question is: does that always happen at some point regardless of the value of x? This is formalized as follows by the *Collatz Conjecture*.

Collatz Conjecture

For any positive integer x, there is $i \geq 0$ such that $x_i = 1$.

The above conjecture is still open, which means there is no proof of it's validity. But it has been verified for x up to 10^{20}.

P9.3.1 Total Stopping Time

The *total stopping time* of x is the number of iterations needed to reach 1; it if formally defined as follows:

The **total stopping time** of x is the smallest i such that $x_i = 1$.

For example, since the orbit of 3 for h is

$$3, 10, 5, 16, 8, 4, 2, 1, \ldots$$

the total stopping time of 3 is 7.

If the Collatz Conjecture is true, then the total stopping time is always a finite number.

P9.3.2 Exercise: Defining $h(x)$

Define a python `h(x)` returns $h(x)$ for a positive integer `x`.

Example of a call to `h`:

```
print(h(11))
```

This should output:

```
34
```

```
#
# Code here
#
```

P9.3.3 Exercise: Defining `totalstoppingtime(x)`

Assume that the positive integer x is less than 10^{20}. Then the Collatz Conjecture is true for x and the total stopping time of x is well-defined.

Define a python function `totalstoppingtime(x)` which returns the total stopping time of `x`.

Example of a call to `totalstoppingtime`:

```
print(totalstoppingtime(3))
```

This should output:

```
7
```

```
#
# Code here
#
```

P9.3.4 A scatter plot from a list of points

The `matplotlib` module can be used to plot graphs. Let us first install and import `matplotlib` first. Installation can be done in a terminal using the command

```
pip3 install -U matplotlib
```

Or in Jupyter by executing:

```
!pip3 install -U matplotlib
```

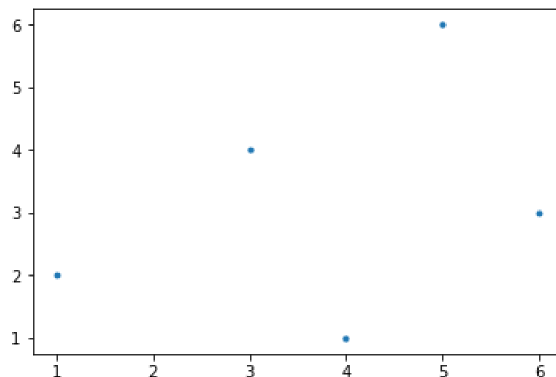
It can then be imported in the code:

```
import matplotlib.pyplot as plt
```

A *scatter plot* is the depiction of a set of points. These points are given in a list of tuples, for example `[(1,2),(3,4),(5,6),(6,3),(4,1)]`.

For these points to be plotted by `matplotlib`, they need to be separated into the x-coordinates and the y-coordinates. In the following code `X` is the list of x-coordinates and `Y` is the list of y-coordinates. The call `plt.plot(X,Y,'.')` defines the plot and `plt.show()` displays it.

```
import matplotlib.pyplot as plt
L=[(1,2),(3,4),(5,6),(6,3),(4,1)]
X=[x for x,y in L]
Y=[y for x,y in L]
plt.plot(X,Y,'.')
plt.show()
```



P9.3.5 Exercise: A scatter plot for the Total Stopping Time

For a positive `x`, the point `(x,totalstoppingtime(x))` is the *stopping point* at `x`.

For example, the list of all stopping points of `x` where $1 \le x \le 8$ is

```
[(1, 0), (2, 1), (3, 7), (4, 2), (5, 5), (6, 8), (7, 16), (8, 3)]
```

Write a program that:

1. Computes the list `S` of all stopping points of `x` where $1 \leq x \leq 100000$.
2. Plots a scatter plot of `S` using `matplotlib`.
3. Find all stopping point(s) in `S` which give(s) the **maximum** total stopping time. Note that there is a `max` function that returns the maximal element of the list given as argument:

```
max([12,3,24,-9,7,-31,6])
```

```
24
```

```
import matplotlib.pyplot as plt
# Defining S
#
# Code here
#
# Plotting S using matplotlib
#
# Code here
#
# Find the maximum in Y, use it to select values that have this maximum␣
↪stopping time
#
# Code here
#
```

Project 10
The Game of Tic-Tac-Toe

Project contents

P10.1 The rules of the game 167
P10.2 Creating the board 167
P10.3 The winning condition 168
P10.4 Playing the game 170

P10.1 The rules of the game

The goal of this project is to implement the game of Tic-Tac-Toe.

The game uses a board and 2 players. The board consists of 9 squares aligned in 3 rows and 3 columns. The squares are initially emtpy. The players alternate in filling in the squares with `X`s and `O`s respectively. With each turn, a player fills only one empty square. To win the game each player must fill either a row, a column, or a digonal with his or her letter. The player that accomplishes this first wins the game. It is possible that the game ends in draw if no player has won when there is no more square to fill.

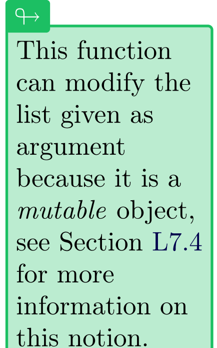
P10.2 Creating the board

The board is stored in a list of size 9. Each element of the list contains one of the following characters:

- `.` a dot if the square is available.
- `X` if player X occupies it.
- `O` if player O occupies it.

Each game starts with an empty board: it must be initialized, namely fill with dots. Here is how this may be done:

```
def initialize(board):
    for i in range(9):
        board += '.' # Add the 9 '.' to the list
    # This function modifies the board but does not return anything

def show(board):
    for i in range(len(board)):
        print('%s ' % board[i], end = ' ')
        if i > 0 and (i+1) % 3 == 0:
```


This function can modify the list given as argument because it is a *mutable* object, see Section L7.4 for more information on this notion.

```
            print('\n',end='') # Print a new line every 3 squares
def main():
    board = []
    initialize(board)
    show(board)

main()
```

```
.  .  .
.  .  .
.  .  .
```

P10.3 The winning condition

After each turn, the program needs to check the status of the board to determine if there is a winner, a draw, or not finished. The `checkwin` function analyzes the board and returns one of the following strings:

- return `'X'` if player X wins;
- return `'O'` is player O wins;
- return `'D'` in case of a draw;
- return `'NF'` in case the game has not finished yet.

To do so:

1. Check the board to see if there is a winner. This is detected as follows:
    - check if any of the 3 rows contains only `X` or `O`;
    - check if any of the 3 columns contains only `X` or `O`;
    - check if any of the 2 diagonals contains only `X` or `O`.

    If any of the above cases applies, then this player has won.
2. Check if the board has empty squares.
    - If that is the case, the game is not finished.
    - Otherwise, the game is a draw.

To test the `checkwin` function, the program asks the user to enter an entire board, namely a string of 9 characters with each character being either `.`, `X`, or `O`. For example: `XOX...X.X`

```
def checkwin(board):
    # Returns 'X' if X wins
    # Returns 'O' if O wins
    # Returns 'D' for draw
    # Returns 'NF' for not finished yet
    # Check winning by rows: 0,1,2; then 3,4,5; then 6,7,8;
    for i in range(3): # i is {0, 1, 2}, so 3*i is {0, 3, 6}
```

```
        if (board[3*i] == board[3*i+1] and board[3*i+1] == board[3*i+2]␣
↪and board[3*i] != '.'):
            # Squares in the row are identical and not '.'
            return board[3*i]
    # Check winning by columns: 0,3,6;  then 1,4,7; then 2,5,8
    for i in range(3): # i is {0, 1, 2}
        if ( board[i] == board[i+3] and board[i+3] == board[i+6] and␣
↪board[i] != '.'):
            # Squares in the column are identical and not '.'
            return board[i]
    # Check winning by the first diagonal: 0,4,8
    if (board[0] == board[4] and board[4] == board[8] and board[0] != '.
↪'):
        # Squares in the diagonal are identical and not '.'
        return board[0]
    # Check winning by the second diagonal: 2,4,6
    if (board[2] == board[4] and board[4] == board[6] and board[2] != '.
↪'):
        # Squares in the diagonal are identical and not '.'
        return board[2]
    # Check for dots (empty squares), i.e. not finished
    for i in range(9):
        if board[i] == '.':
            return 'NF'
    # If the code reached this far without returning, then it is a draw
    return 'D'

def main():
    while True:
        board = []
        initialize(board)
        s = input('Press q to quit or a 9 character string (use only␣
↪`X`,`O`,`.`) : ')
        if s == 'q':
            break
        for i in range(min(len(s),9)):
            board[i] = s[i] # Set the board status from the string
        show(board)
        result = checkwin(board)
        if result == 'D':
            print('The game ends in a draw.')
        elif result == 'NF':
            print('The game has not finished.')
        else:
            print('Player {} wins the game.'.format(result))
        print('--------------------------')
```

```
main()
```

```
Press q to quit or a 9 character string (use only `X`,`O`,`.`) : ...
.  .  .
.  .  .
.  .  .
The game has not finished.
--------------------------
Press q to quit or a 9 character string (use only `X`,`O`,`.`) : ...XO..OX
.  .  .
.  X  O
.  .  O
The game has not finished.
--------------------------
Press q to quit or a 9 character string (use only `X`,`O`,`.`) : OX..OX..O
O  X  .
.  O  X
.  .  O
Player O wins the game.
--------------------------
Press q to quit or a 9 character string (use only `X`,`O`,`.`) : XOXOXOOXO
X  O  X
O  X  O
O  X  O
The game ends in a draw.
--------------------------
Press q to quit or a 9 character string (use only `X`,`O`,`.`) : q
```

P10.4 Playing the game

The function that plays the game consists of an indeterminate `while` loop. Each iteration consists of the following steps:

1. Player X is asked to enter a board square as a number 0 .. 8; this process is repeated until the player enters a valid number representing an empty square.
2. The square is filled with `'X'`.
3. The board is shown and the `checkwin` function is called; in case of a winner, the function returns the either `'X'` or `'O'`.
4. Player O is asked to enter a board square as a number 0 .. 8; this process is repeated until the player enters a valid number representing an empty square.
5. The square is filled with `'O'`.
6. The board is shown and the `checkwin` function is called; in case of a winner, the function returns the either `'X'` or `'O'`.

The `main` function allows players to play the several rounds of the game. It consists of a `while` loop as follows:

1. Initialize the board (empty the squares)

2. Ask the user to either quit or play the game
3. The game is played and the result is printed

```
def playgame(board):
    while True:
        # Ask player X for a board spot
        while True:
            i = input('Player X: enter a board spot between 0 and 8: ')
            if i.isdigit() == False: # Input is not a number
                print('Invalid input. Try again.')
                continue
            i = int(i)
            if i < 0 or 8 < i:        # Input is not between 0 and 8
                print('Invalid input. Try again.')
                continue
            if board[i] != '.':
                print('Board occupied. Try again.')
                continue
            else:
                board[i] = 'X'        # Set this square to 'X'
                break                 # Stop the loop
        # Check board status
        show(board)
        if checkwin(board) != 'NF':
            return checkwin(board)

        # Ask player O for a board spot
        while True:
            i = input('Player O: enter a board spot between 0 and 8: ')
            if i.isdigit() == False: # Input is not a number
                print('Invalid input. Try again.')
                continue
            i = int(i)
            if i < 0 or 8 < i:        # Input is not between 0 and 8
                print('Invalid input. Try again.')
                continue
            if board[i] != '.':
                print('Board occupied. Try again.')
                continue
            else:
                board[i] = 'O'        # Set this square to 'O'
                break                 # Stop the loop
        # Check board status
        show(board)
        if checkwin(board) != 'NF':
            return checkwin(board)

def main():
```

```
    while True:
        board = []
        initialize(board)
        show(board)
        s = input('Press q to quit or anything else to play TIC-TAC-TOE:␣
↪')
        if s == 'q':
            break
        result = playgame(board)
        if result == 'D':
            print('The game ends in a draw.')
        else:
            print('Player {} wins the game.'.format(result))

main()
```

```
.  .  .
.  .  .
.  .  .
Press q to quit or anything else to play TIC-TAC-TOE: Ok
Player X: enter a board spot between 0 and 8: 2
.  .  X
.  .  .
.  .  .
Player O: enter a board spot between 0 and 8: 3
.  .  X
O  .  .
.  .  .
Player X: enter a board spot between 0 and 8: 1
.  X  X
O  .  .
.  .  .
Player O: enter a board spot between 0 and 8: 4
.  X  X
O  O  .
.  .  .
Player X: enter a board spot between 0 and 8: 5
.  X  X
O  O  X
.  .  .
Player O: enter a board spot between 0 and 8: 6
.  X  X
O  O  X
O  .  .
Player X: enter a board spot between 0 and 8: 9
Invalid input. Try again.
Player X: enter a board spot between 0 and 8: 8
.  X  X
```

```
0  0  X
0  .  X
Player X wins the game.
.  .  .
.  .  .
.  .  .
Press q to quit or anything else to play TIC-TAC-TOE: Ok
Player X: enter a board spot between 0 and 8: 4
.  .  .
.  X  .
.  .  .
Player O: enter a board spot between 0 and 8: 0
0  .  .
.  X  .
.  .  .
Player X: enter a board spot between 0 and 8: 1
0  X  .
.  X  .
.  .  .
Player O: enter a board spot between 0 and 8: 7
0  X  .
.  X  .
.  0  .
Player X: enter a board spot between 0 and 8: 3
0  X  .
X  X  .
.  0  .
Player O: enter a board spot between 0 and 8: 5
0  X  .
X  X  0
.  0  .
Player X: enter a board spot between 0 and 8: 5
Board occupied. Try again.
Player X: enter a board spot between 0 and 8: 2
0  X  X
X  X  0
.  0  .
Player O: enter a board spot between 0 and 8: 6
0  X  X
X  X  0
0  0  .
Player X: enter a board spot between 0 and 8: 8
0  X  X
X  X  0
0  0  X
The game ends in a draw.
.  .  .
.  .  .
```

```
. . .
Press q to quit or anything else to play TIC-TAC-TOE: q
```

Project 11
A Function-Based Role Playing Game

Project contents

P11.1 Introduction **175**
P11.2 Program structure **175**
P11.2.1 User choices 175
P11.2.2 One room = one function 176
P11.2.3 The hero status 177
P11.2.4 A complete mini-donjon 177
P11.3 Let's build a game! **179**
P11.3.1 The donjon map 179
P11.3.2 Coding phase 1: The donjon structure 179
P11.3.3 Coding phase 2: The room functions 182
P11.3.4 Possible extensions 192

P11.1 Introduction

The goal of this project is to implement a small *Role Playing Game* (RPG), where the user plays a hero visiting a donjon.

A donjon is made of several rooms, and different things happen whenever the hero enters the room: fight a monster, find a treasure, etc.

Throughout the game, the program must remember some information about the hero: his name, his number of points of life, his number of coins. This status information could be extended to add more features to the game.

P11.2 Program structure

P11.2.1 User choices

The player is given the opportunity to make choices by inputting some text. It is a good idea for these games to be case-insensitive, so to convert the input to lowercase before testing it against other lowercase value. So in the example below, it does not matter if the player enters "right", "Right", "RIGHT", "rIgHt", or any variation of letter case, they would all be converted to "right" before comparison. This test is done in a conditional: does the input value match the choices that were offered? Since there may be more that one possibility, the `elif` (else-if) construct is used.

For a refresher on conditions, see Lecture L5.

```
print("Do you want to go left, right, or straight? (Enter \"left\",␣
↪\"right\", or \"straight\")")
userIn=input().lower() # lowercase of input so case does not matter
if userIn=="left":     # compare to lowercase version; remark the ==␣
↪(double equal)
    print("Let's go to the left.")
elif userIn=="right":
    print("Right it is, then.")
elif userIn=="straight":
    print("Straight ahead!")
else:
    print("Invalid input. You lose, game over!")
```

```
Do you want to go left, right, or straight? (Enter "left", "right", or␣
↪"straight")
RIGHT
Right it is, then.
```

P11.2.2 One room = one function

For a refresher on functions, see Lecture L8.

The code of this project will be divided into several functions, following the principle that *one donjon room is coded by one function*. Whatever happens inside this room is coded in that function, and moving to another room is performed by calling the function corresponding to that room.

That means all functions have to be defined before a call can actually happen. While coding, it is possible to just define the function but leave the body empty.

For example, see this incomplete 2-room donjon:

```
def entranceRoom():
    print("You enter the room... Nothing happens.")
    print("There is a door to the right. Do you want to enter it?")
    print("(Enter \"yes\" to enter and anything else to exit)")
    userIn=input().lower() # lowercase of input so case does not matter
    if userIn=="yes":
        print("You leave by the door to the right...")
        rightRoom()
    else:
        print("You leave the donjon, it is the end of the adventure!")

def rightRoom():
    # TO DO!
    return
entranceRoom()
```

```
You enter the room... Nothing happens.
There is a door to the right. Do you want to enter it?
(Enter "yes" to enter and anything else to exit)
```

```
stop
You leave the donjon, it is the end of the adventure!
```

To be able to remember the status as the hero goes through the donjon rooms, the hero status is passed as an **argument** to the next room.

P11.2.3 The hero status

The hero status is a *dictionary*. It is a list that is indexed by words instead of numbers from 0 to size-1. It is possible to add new values to the dictionary simply by setting the value at a new index. Indices will be added to this dictionary as is necessary to code the game.

For a more thorough introduction to dictionaries, see Lecture L7.3.

```
hero = {'name':"Sigsegv",'life':100,'gold':0}  # Original hero
print(hero)
print('magic' in hero.keys())
hero['gold'] = hero['gold']+10                 # Add 10 gold
hero['magic'] = 42                             # New field: 'magic', with␣
↪value 42
print(hero)
```

```
{'name': 'Sigsegv', 'life': 100, 'gold': 0}
False
{'name': 'Sigsegv', 'life': 100, 'magic': 42, 'gold': 10}
```

P11.2.4 A complete mini-donjon

```
def entranceRoom(h):
    print("You enter the room... There is a goblin!")
    print("Do you use your sword or punch him?")
    print("(Enter \"Sword\" to use the sword and \"Punch\" to punch the␣
↪goblin)")
    userIn = input().lower() # lowercase of input so case does not matter
    inputOk = False
    if userIn=="sword":
        inputOk = True
        print("You killed the goblin with minor injuries. You lose 5 life␣
↪points and gain 20 coins.")
        h['life'] = h['life']-5
        h['gold'] = h['gold']+20
    elif userIn=="punch":
        inputOk = True
        print("You killed the goblin, but it got you badly! You lose 50␣
↪life points and gain 20 coins.")
        h['life'] = h['life']-50
        h['gold'] = h['gold']+20
    else: # invalid input
        print("The input was invalid. Let's try again")
        entranceRoom(h) # re-enter the same room
```

```
    if inputOk: # In case the input was incorrect, reenter the same room
        if h['life']<=0: # No more life points
            dead(h)
        else:
            print("There is a door to the right. Do you want to enter it?
↪")
            print("(Enter \"Yes\" to enter and anything else to exit)")
            userIn = input().lower() # lowercase of input so case does␣
↪not matter
            if userIn=="yes":
                print("You leave by the door to the right...")
                rightRoom(h)
            else:
                print("You leave the donjon, it is the end of the␣
↪adventure!")
                printStats(h)

def rightRoom(h):
    print("There is a treasure. You gain 100 coins")
    h['gold'] = h['gold']+100
    print("There is nothing else, so you can only go back from where you␣
↪came.")
    entranceRoom(h)
    return

def dead(h):
    print("You died, game over!")
    printStats(h)

def printStats(hero):
    print("Your name was %s, you had %d life points and owned %d gold␣
↪coins." % (hero['name'],hero['life'],hero['gold']))

def game():
    hero = {'name':"Sigint",'life':100,'gold':0}
    entranceRoom(hero)
game()
```

```
You enter the room... There is a goblin!
Do you use your sword or punch him?
(Enter "Sword" to use the sword and "Punch" to punch the goblin)
Punch
You killed the goblin, but it got you badly! You lose 50 life points and␣
↪gain 20 coins.
There is a door to the right. Do you want to enter it?
(Enter "Yes" to enter and anything else to exit)
stop
```

```
You leave the donjon, it is the end of the adventure!
Your name was Sigint, you had 50 life points and owned 20 gold coins.
```

P11.3 Let's build a game!

P11.3.1 The donjon map

It is important to make a plan in advance: decide how rooms communicate with another, and maybe write down what's going to happen in each room.

In Figure P11.1 is a map of the suggested donjon. Both the corridor and the cellar can be removed to simplify the game. Doors have different colors (wooden, metal, gilded) so that the player can distinguish them.

Let's briefly describe what will happen in each room.

- Entrance: nothing happens (could be added later); exit is possible, but means death (game over)
- Guards room: a goblin is there, choice of sword (-5L,+20G) or punch (-50L,+20G)
- Corridor (optional): encounter an orc, must do punch-sword sequence to kill them: otherwise -10L per turn and back to entrance after 2 turns ; killing the orc brings +15L
- Cellar (optional): an old man is hidden in there; you can play a riddle game for 5G, if answer is correct you earn 20G.
- Ball room: nothing happens (left to the imagination of the individual programmer)
- Kitchen: can eat some food to replenish life points up to max:
- ham: +15L
- cheese: +10L
- bread: +5L
- beer: +20L, -5G (coins dropped out of drunkenness)
- Throne room: the final boss arrives (-10L) to protect the Crown of Fame. Must do sword-punch-sword sequence to kill the boss, while getting hurt (-15L) each turn; after 4 turns, back to the kitchen. When the boss is dead, gets the Crown of Fame and wins the game.

P11.3.2 Coding phase 1: The donjon structure

First, all the functions for all the rooms must be defined, leaving their body empty for now. Comments can describe what should happen and what are the connecting rooms to facilitate the coding of the body of the function. A function for game over must also be defined, with a boolean argument to determine whether the hero finished the game or died.

Once all functions exist, the game can be "played" through a testing function that initializes the hero dictionary and have him enter the donjon.

```python
#### After phase 1

def entrance(hero):
    # Entrance
    # Scenario TBD
```

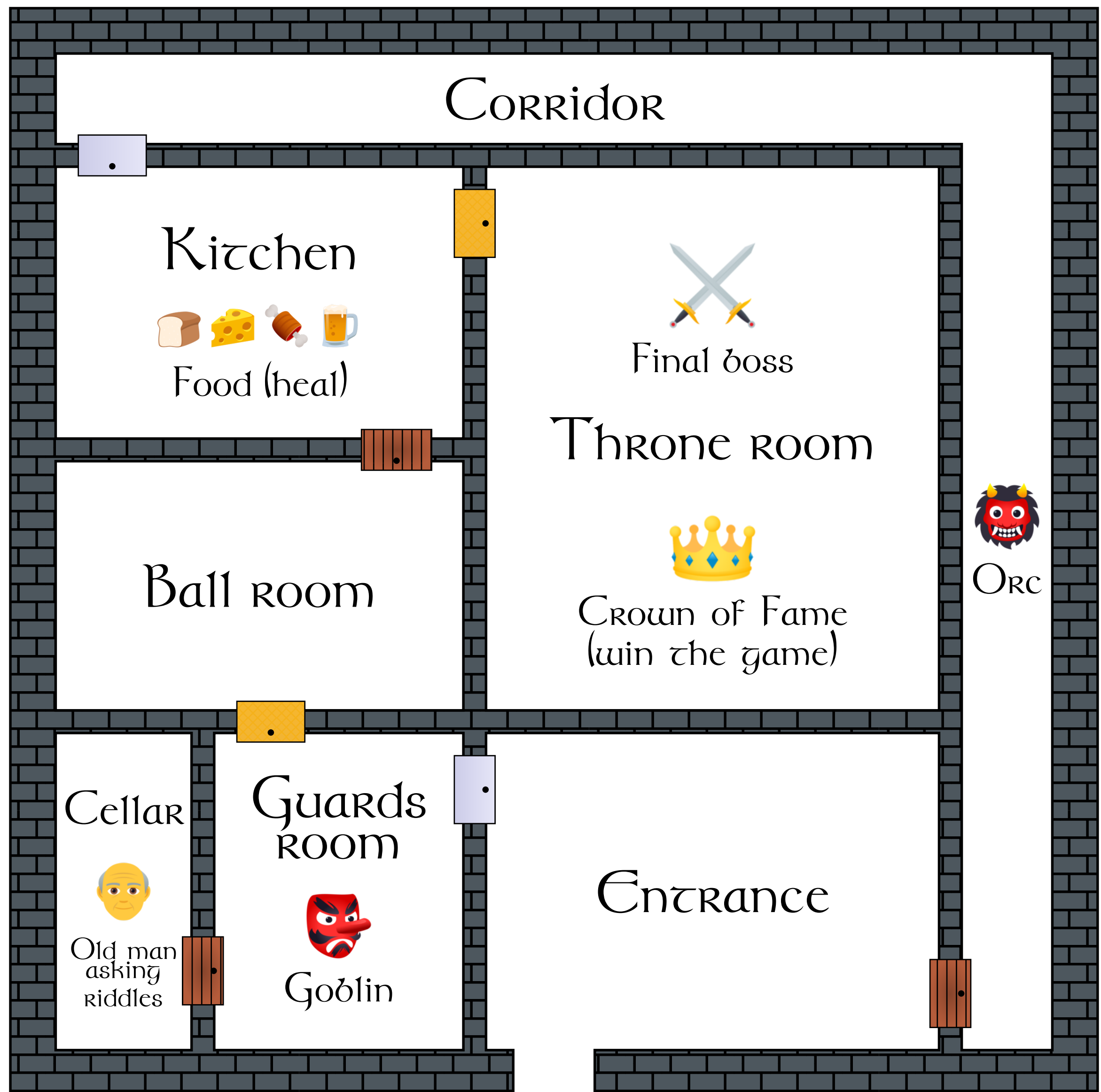


Figure P11.1: Map of the suggested donjon

```
    # Door to the left, metal: guards room (function call␣
↪guardsRoom(hero))
    # Door to the right, wood: corridor (function call corridor(hero))
    # Exit is possible, but means death
    return # one instruction is mandatory for correct syntax

def guardsRoom(hero):
    # Guards room
    # A goblin is there, choice of sword (-5L,+20G) or punch (-50L,+20G)
    # Door to the front, wooden: cellar (function call cellar(hero))
    # Door to the back, metal: entrance (function call entrance(hero))
    # Door to the right, gilded: ball room (function call ballroom(hero))
    return

def corridor(hero):
    # Corridor
    # Encounter an orc, must do punch-sword sequence to kill them:␣
↪otherwise -10L per turn and back to entrance after 2 turns.
    # Killing the orc brings +15L
    # Door at the start: entrance (function call entrance(hero))
    # Door at the end: kitchen (function call kitchen(hero))
    return

def cellar(hero):
    # Cellar
    # An old man is hidden in there. You can play a riddle game for 5G.␣
↪If answer is correct, earn 20G.
    # Door at the back, wood: guards room (function call guardsRoom(hero))
    return

def ballroom(hero):
    # Ball room
    # Scenario TBD
    # Door to the back, gilded: guards room (function call␣
↪guardsRoom(hero))
    # Door to the front, wood: kitchen (function call kitchen(hero))
    return

def kitchen(hero):
    # Kitchen
    # Can eat some food to replenish life points up to max
    #  * ham: +15L
    #  * cheese: +10L
    #  * bread: +5L
    #  * beer: +20L, -5G (coins dropped out of drunkenness)
    # Door to the back, wood: ball room (function call ballroom(hero))
    # Door to the front, metal: corridor (function call corridor(hero))
```

```
    # Door to the right, gilded: throne room (function call throne(hero))
    return

def throne(hero):
    # Throne room
    # The final boss arrives (-10L) to protect the Crown of Fame.
    # Must do sword-punch-sword sequence to kill the boss.
    # Each turn -15L, after 4 turns, back to the kitchen.
    # When the boss is dead, gets the Crown of Fame and wins the game.
    return

def gameOver(hero,win=False):
    # Simple function, will revisit later
    if win:
        print("You won!")
    else:
        print("You died!")
    return

def testing():
    h = {'name':"Sigsys",'life':100,'gold':0,'max':100} # We need max to␣
↪know how much life he can have
    print("Testing of the game, initial hero:",h)
    entrance(h)

testing() # So far, only prints the hero dictionary.
```

```
Testing of the game, initial hero: {'name': 'Sigsys', 'max': 100, 'life':␣
↪100, 'gold': 0}
```

P11.3.3 Coding phase 2: The room functions

All the room functions can now be filled with actual code to incorporate what happens in the room.

Auxiliary functions

Some tasks that are performed often can be put in a function. That function will be called instead of repeating the code. This is the case of gaining (or losing) life points and gold coins.

Both these functions check that the values respect certain bounds.

```
def life(hero,v):
    if v<0:
        if (hero['life']+v<0): # No negative life points.
            hero['life'] = 0
        else:
            hero['life'] = hero['life']+v
```

```
        print("  \U0001FA78 You lose %d life points\t\t[%d/%d]" %␣
↪(-v,hero['life'],hero['max'])) # Blood emoji
    else:
        if (hero['life']+v>hero['max']): # Not higher than max.
            hero['life'] = hero['max']
        else:
            hero['life'] = hero['life']+v
        print("  \U00002695 You gain %d life points\t\t[%d/%d]" %␣
↪(v,hero['life'],hero['max'])) # Medecine emoji
    return hero

def gold(hero,v):
    if v<0:
        if (hero['gold']+v<0): # No negative money. This should be tested␣
↪before calling this function.
            hero['gold'] = 0
        else:
            hero['gold'] = hero['gold']+v
        print("  \U0001FA99 You lose %d gold coins\t\t[%d]" %␣
↪(-v,hero['gold'])) # Coin emoji
    else:
        hero['gold'] = hero['gold']+v
        print("  \U0001F4B0 You gain %d gold coins\t\t[%d]" %␣
↪(v,hero['gold'])) # Bag of money emoji
    return hero
```

About Emojis

Emojis may not show on the terminal or browser you are using.

For more information about emojis in Python, see Section L4.5.2.

The entrance

This function will take some input from the player in order to know which room to be visited next. If he exits, he dies a ridiculous death (for example a stone falls on his head). For any invalid input, the user can try again by calling the very same `entrance` function.

Exercise The function has been partially written; complete the code so that the function reacts accordingly to user input.

```
def entrance(hero):
    # Entrance
    # Scenario TBD
    # Door to the left, metal: guards room (function call␣
↪guardsRoom(hero))
    # Door to the right, wood: corridor (function call corridor(hero))
    # Exit is possible, but means death
    print("You are in the entrance hall.")
    # So far nothing really happens in here, just a choice of doors.
    print("There is a wooden door and a metal door.")
```

```
    print("[Enter \"Wooden\" or \"Metal\" to choose a door, or \"Exit\"␣
↪to leave the donjon (and the game)]")
    #
    # Complete the code here so that the function reacts accordingly to␣
↪user input
    #
    return
```

The guards room

In this function, user input will be first used to decide how the hero will fight. When the fight is over (and if the hero is not dead) the hero gains 20 pieces of gold and the player is asked to choose a door.

The code below only contains the first phase of the action in this room (the fight). Note that the second phase should only happen if the input was valid, which is kept in the boolean variable `inputOk`.

Exercise Complete the second phase of the function. It is assumed than an invalid choice of door makes the hero reenter the same room (by calling the `guardsRoom` function).

```
def guardsRoom(hero):
    # Guards room
    # A goblin is there, choice of sword (-5L,+20G) or punch (-50L,+20G)
    # Door to the front, wooden: cellar (function call cellar(hero))
    # Door to the back, metal: entrance (function call entrance(hero))
    # Door to the right, gilded: ball room (function call ballroom(hero))
    print("You are in the guards room. A goblin appears!")
    print("Do you use your sword or punch him?")
    print("[Enter \"Sword\" to use the sword and \"Punch\" to punch the␣
↪goblin]")
    i = input().lower() # lowercase of input so case does not matter
    inputOk=False
    if i=="sword":
        inputOk = True
        print("\U0001F5E1 You take your sword out. The goblin jumps at␣
↪you with claws out and impales himself on your blade.")
                # Sword emoji
        print("You killed the goblin with only some scratches.")
        hero=life(hero,-5)
    elif i=="punch":
        inputOk=True
        print("\U0001F44A The goblin jumps at you with claws out teeth␣
↪bare. You punch at him until it finally dies, but you are badly hurt.")
                # Punch emoji
        hero = life(hero,-50)
    else: # invalid input
        print("The input was invalid. Let's try again")
```

```
        guardsRoom(hero) # re-enter the same room
    if inputOk: # In case the input was incorrect we don't want to␣
↪continue, but we reenter the same room
        #
        # Complete the second phase of the function here
        #
    return
```

The corridor

In this, the hero has to punch then use his sword. This can be implemented by nested `if` conditionals. Any invalid input or the wrong decision and he is sent back to the entrance hall having lost 10 life points (if the hero is not dead).

Exercise Complete the `corridor` function.

```
def corridor(hero):
    # Corridor
    # Encounter an orc, must do punch-sword sequence to kill them:␣
↪otherwise -10L per turn and back to entrance after 2 turns.
    # Killing the orc brings +15L
    # Door at the start, wood: entrance (function call entrance(hero))
    # Door at the end, metal: kitchen (function call kitchen(hero))
    return
```

The cellar

In this function, nested `if` conditionals once again code the successive choices of the player:

- Chooses to play:
- Not enough money: is sent back to the guards room
- Enough money:
  - Correct answer: gets 20 pieces of gold, is sent back to the guards room
  - Incorrect answer: reenters the cellar (try again)
- Chooses not to play: is sent back to the guards room
- Invalid input: reenter the cellar (try again)

Exercise

1. Devise a riddle, preferably with an answer that is a single word.
2. Complete the `cellar` function defined in Section P11.3.2 to incorporate all the user choices, using the riddle you wrote.

The Ball room

In this room nothing happens; that could be changed later on. The basic version below only has ways to move out of it.

```
def ballroom(hero):
    # Ball room
    # Scenario TBD
    # Door to the back, gilded: guards room (function call␣
↪guardsRoom(hero))
    # Door to the front, wood: kitchen (function call kitchen(hero))
    print("You are in a nice ballroom. There is a gilded door and a door␣
↪wooden door.")
    print("[Enter \"Gilded\" or \"Wooden\"]")
    i = input().lower()
    if i=="gilded":
        guardsRoom(hero)
    elif i=="wooden":
        kitchen(hero)
    else:
        print("You take a tour of the room.")
        ballroom(hero)
    return
```

The kitchen

The kitchen offers several options that heal the hero in preparation for the final boss. This function is in two phases:

- First a choice of food; invalid input meaning no food at all.
- Then a choice of door; invalid input meaning staying reentering the same room (for example having a pot fall on the hero's head, incurring a loss of 1 life point before waking up again).

```
def kitchen(hero):
    # Kitchen
    # Can eat some food to replenish life points up to max
    #   * ham: +15L
    #   * cheese: +10L
    #   * bread: +5L
    #   * beer: +20L, -5G (coins dropped out of drunkenness)
    # Door to the back, wood: ball room (function call ballroom(hero))
    # Door to the front, metal: corridor (function call corridor(hero))
    # Door to the right, gilded: throne room (function call throne(hero))
    print("You are in a kitchen. Food smells good, there is ham, bread,␣
↪cheese, and beer; which do you choose?")
    print("[Enter \"Ham\", \"Bread\", \"Cheese\", or \"Beer\"]")
    i = input().lower()
    if i=="ham":
        print("\U0001F356  \"Humm, what a delicious ham!\"")
        hero = life(hero,15)
    elif i=="bread":
```

```
        print("\U0001F35E  \"A crispy loaf of bread!\"")
        hero = life(hero,5)
    elif i=="cheese":
        print("\U0001F9C0  \"What a nice piece of cheese!\"")
        hero = life(hero,10)
    elif i=="beer":
        print("\U0001F37A  \"Nice fresh beer... I do feel tipsy, though...
↪\"")
        hero = life(hero,20)
        hero = gold(hero,-5)
    else: # invalid input
        print("  \"Nah, I'm good\"")
    print("After this little snack, you take a look around.")
    print("There are three doors: one wooden, one gilded, and one in␣
↪metal.")
    print("[Enter \"Wooden\", \"Gilded\" or \"Metal\" to choose a door]")
    i = input().lower()
    if i=="wooden":
        ballroom(hero)
    elif i=="metal":
        corridor(hero)
    elif i=="gilded":
        throne(hero)
    else:
        print("You rummage through the food stores. A pot falls on your␣
↪head.")
        hero = life(hero,-1)
        if hero['life']<=0:
            gameOver(hero,win=False)
        else:
            print("You wake up.")
            kitchen(hero)
    return
```

The throne room: the final boss

In this room, it is possible to have nested conditionals to simulate the 4 turns of battle, but that means at least 16 cases! It may be difficult to keep track of them. Since all that matters is the input at previous turns, one function per turn will be used, using the inputs of previous turns as arguments.

A function to be called when the boss has been defeated is also needed. This is because in 4 turns, there are two ways to produce the sequence sword-punch-sword that kills the boss: sword-punch-sword- or punch-sword-punch-sword.

```
def turn1(hero):
    return
```

```
def turn2(hero,t1):
    # Gets the choice of turn 1: 1 for sword, -1 for punch, 0 otherwise
    return

def turn3(hero,t1,t2):
    # Gets the choices for turns 1 and 2: 1 for sword, -1 for punch, 0 otherwise
    return

def turn4(hero,t1,t2,t3):
    # Gets the choices for turns 1, 2, and 3: 1 for sword, -1 for punch, 0 otherwise
    return

def victory(hero):
    # The boss has been killed
    # Get the Crown of Fame and win the game.
    return

def throne(hero):
    # Throne room
    # The final boss arrives (-10L) to protect the Crown of Fame.
    # Must do sword-punch-sword sequence to kill the boss.
    # Each turn -15L, after 4 turns, back to the kitchen.
    # When the boss is dead, gets the Crown of Fame and wins the game.
    print("You enter the throne room. You see the Crown of Fame glowing on a cushion next to the throne.")
    print("As you walk to grab it, a deep voice resonates through the room.")
    print("\"I have been expecting you, %s. You are not the first adventurer who tries to steal my Crown." % hero['name'])
    print("And you won't be the last. You will be just another pile of bones!\"")
    print("Suddenly, a knight in black armour falls from the ceiling and slashes at you with his flail.")
    hero = life(hero,-10)
    if hero['life']>0:
        turn1(hero)
    else:
        gameOver(hero,win=False)
    return
```

The functions for the 4 turns of combat need to be completed. Any invalid choice of weapon means the hero does nothing and suffers the boss' strike.

Exercise Complete the functions for turns 2 and 4 below.

```
def turn1(hero):
    print("Do you use your sword or punch him?")
    print("[Enter \"Sword\" to use the sword and \"Punch\" to punch the␣
↪dark knight]")
    i = input().lower()
    choice = 0
    if i=="sword": # Correct choice here
        choice = 1
        print("\U0001F5E1 You slash at the knight's side. He blocks with␣
↪his shield and sends a kick from below it.")
    elif i=="punch":
        choice = -1
        print("\U0001F44A You try to punch the knight. He blocks with his␣
↪shield. The shield is of steel, your hand is of flesh. Guess who wins...
↪")
    else:
        print("You are stunned by the knight's apparition. He takes␣
↪advantage of that to give you a second dose of flail.")
    hero = life(hero,-15)
    if hero['life']>0:
        turn2(hero,choice)
    else:
        gameOver(hero,win=False)
    return
```

```
def turn2(hero,t1):
    # Gets the choice for turn 1: 1 for sword, -1 for punch, 0 otherwise
    #
    # Complete the second combat turn here
    #
    return
```

```
def turn3(hero,t1,t2):
    # Gets the choices for turns 1, and 2: 1 for sword, -1 for punch, 0␣
↪otherwise
    print("Do you use your sword or punch him?")
    print("[Enter \"Sword\" to use the sword and \"Punch\" to punch the␣
↪dark knight]")
    i = input().lower()
    choice = 0
    if i=="sword":
        choice = 1
        if (t1==1 and t2==-1): #Correct sequence
            print("\U0001F5E1 You slash at the knight's head and your␣
↪blade gets right between breastplate and helmet. The head rolls to the␣
↪floor.")
```

```
            victory(hero)
        else:
            print("\U0001F5E1 You slash at the knight's head, but the␣
↪dark knight ducks and shoves his shield in your belly.")
            hero = life(hero,-15)
            if hero['life']>0:
                turn4(hero,t1,t2,choice)
            else:
                gameOver(hero,win=False)
    elif i=="punch":
        choice = -1
        print("\U0001F44A You punch the knight. He spins around but␣
↪extends his leg in the process to land a roundhouse kick to your chest.
↪")
        hero = life(hero,-15)
        if hero['life']>0:
            turn4(hero,t1,t2,choice)
        else:
            gameOver(hero,win=False)
    else:
        print("You are overwhelmed by the knight's attacks and can't␣
↪decide. Your defense is wide open for another dose of flail.")
        hero = life(hero,-15)
        if hero['life']>0:
            turn4(hero,t1,t2,choice)
        else:
            gameOver(hero,win=False)
    return
```

```
def turn4(hero,t1,t2,t3):
    # Gets the choices for turns 1, 2, and 3: 1 for sword, -1 for punch,␣
↪0 otherwise
    #
    # Complete the fourth combat turn here
    #
    return
```

Now for the `victory` function:

```
def victory(hero):
    # The boss has been killed
    # Get the Crown of Fame and win the game.
    print("The dark knight is dead. You catch your breath as you walk␣
↪slowly to the Crown of Fame.")
    print("You take the Crown in your fingertips and place it on your␣
↪head.\n  \U0001F451 You made it, you are famous!")
```

```
    print("There will be song sung in your glory for the centuries ahead.␣
↪Unless an adventurer manages to steal the Crown from you!")
    gameOver(hero,win=True)
    return
```

A fancier version of the end of game function

```
def gameOver(hero,win=False):
    if win:
        print("\t\tCongratulations, you finished the game!")
    else:
        print("\t\tYou died! Game Over!")
    print("\t\U0001F3B6 O %s, mighty adventurer, you had %d life points␣
↪and owned %d gold coins... \U0001F3B5" %␣
↪(hero['name'],hero['life'],hero['gold']))
    # \U0001F3B6 and \U0001F3B5 are music notes emojis.
    return
```

Testing function

To call directly the function of the room to be tested, it is easier to have a testing function:

```
def testing():
    h = {'name':"Sigsys",'life':100,'gold':0,'max':100} # max is used to␣
↪know how much life the hero can have
    print("Testing of the game, initial hero:",h)
    #entrance(h)
    #guardsRoom(h)
    #cellar(h)
    #ballroom(h)
    #kitchen(h)
    #throne(h)
    return

testing()
```

```
Testing of the game, initial hero: {'name': 'Sigsys', 'max': 100, 'life':␣
↪100, 'gold': 0}
```

The main function for the game

This function creates the initial hero dictionary using a user-provided hero name and have the hero enter the donjon.

```
def donjonGame():
    print("\t\t*** Welcome to this RPG ***\n\t(Real Python Game or Role␣
↪Playing Game)")
    print("Please enter the name of your hero: ")
```

```
    name = input().title() # Capitalize every word
    h = {'name':name,'life':100,'gold':0,'max':100}
    print("Good day, %s!" % name)
    print("You seem ready to go on an adventure. You have your sword and␣
↪a strong fist.\nI suggest you use these in alternance, it usually works␣
↪best against ennemies.")
    print("This mysterious donjon is rumoured to harbour the Crown of␣
↪Fame, which makes his bearer famous.\nIsn't that the goal of any␣
↪adventurer?")
    print("Let's go!")
    entrance(h)
    return
```

```
donjonGame()
```

```
                *** Welcome to this RPG ***
        (Real Python Game or Role Playing Game)
Please enter the name of your hero:
Link
Good day, Link!
You seem ready to go on an adventure. You have your sword and a strong␣
↪fist.
I suggest you use these in alternance, it usually works best against␣
↪ennemies.
This mysterious donjon is rumoured to harbour the Crown of Fame, which␣
↪makes his bearer famous.
Isn't that the goal of any adventurer?
Let's go!
You are in the entrance hall.
There is a wooden door and a metal door.
[Enter "Wooden" or "Metal" to choose a door, or "Exit" to leave the donjon␣
↪(and the game)]
Exit
As you cowardly exit the donjon, a stone that was dislodged by the years␣
↪falls onto your head.
 🩸 You lose 100 life points            [0/100]
                You died! Game Over!
        🎶 O Link, mighty adventurer, you had 0 life points and owned 0 gold
coins... 🎵
```

P11.3.4 Possible extensions

This small game can be easily extended to be fancier. Some simple ideas include:

- Adding some progress indicators in the hero's dictionary to track what happened before. For example it could be used to prevent playing the riddle game after having won it once.

- Using `while` loops to ensure only valid input is entered (and asking for new input while incorrect)
- Using timers to have text appear after a certain number of seconds instead of all at once.

Project 12
Voting Systems

Project contents

P12.1 Introduction **195**
P12.2 The setting, formally **196**
P12.2.1 A note on factorial 196
P12.3 The Python setting **197**
P12.3.1 `int2char` 197
P12.3.2 `genAllPreferences` and `genAllPreferencesFromList` 198
P12.3.3 `genRandomPreferences` 198
P12.3.4 `print` and `prettyPrintPreferences` 199
P12.3.5 Other vote results for testing 200
P12.4 Voting systems **200**
P12.4.1 Preferred ordering 200
P12.4.2 First past the post 203
P12.4.3 The Borda count method 205
P12.4.4 The Condorcet method 205
P12.4.5 Two round voting 207
P12.4.6 Exercise 207
P12.4.7 Elimination voting 208
P12.4.8 Comparison of the voting methods (Exercise) 210

P12.1 Introduction

Voting is a process for a group of individuals to make decision that reflect ***the choice of the majority***. Whether it is a group of people trying to decide where they will have dinner or a whole country electing its leader, individuals each have their preferences (for example: "I prefer Chinese to Italian to Mexican to Indian"). When there are more than two options, there are actually several ways to chose the proposition "preferred by the majority" based on each individual preferences.

All these methods aim at preserving some fairness in the process. Said fairness takes several forms, for example: can voters have an interest in lying about their preference? This is called *strategic voting*, and is seen as a negative trait of the system because it encourages dishonesty. Since there is no mathematically perfect system (one that satisfies all aspects of fairness), the question of which system is best or most fair is often more philosophical than mathematical.

This project is an overview of voting methods. The goal is, for several such systems, to create a program that computes the winner based on individual preferences. The vote is assumed to be performed as follows:

- Each individual has an order of preferences between the options offered.
- There can be no equality in an individual preference ("I like Chinese and Indian equally" is not allowed).
- All options have to be sorted ("I don't even want to consider Italian or Indian" is not allowed).
- There is at most one winner (only one position to be filled).

In some cases the voting system will not be able to discriminate between several options, but that's OK (in country-wide elections, that is very unlikely; in smaller assemblies there might be rules to break such equalities, for example declaring the winner based on seniority).

Many more voting systems exist, especially when the above restrictions are lifted.

P12.2 The setting, formally

There are k candidates to the election (presidents or restaurants, it does not matter). Each is represented by a letter `A`, `B`, `C`...

Each voter has a preference order which is a string made of k letters in the order they prefer: `CABD` means `C` is the preferred choice, `A` is second-most preferred, then `B`, and `D` is the least preferred choice. Note that there are $k!$ (*factorial of* k, see note P12.2.1 below) possible orderings when there are k candidates.

The result of the election is, for each of these ordering, the number of voters that have this particular ordering as their preference relation.

In mathematical terms, an ordering containing all the options is called a *permutation*.

P12.2.1 A note on factorial

The factorial of k, written $k!$, is defined as

$$k! = \prod_{i=1}^{k} i = 1\times 2\times\cdots\times(k-1)\times k = k\times(k-1)\times(k-2)\times\cdots\times 2\times 1 = \begin{cases} 1 & \text{if } k \leq 1 \\ k \times (k-1)! & \text{if } k > 1 \end{cases}$$

This function grows very fast as k grows. For example we have:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 \\ 3! &= 6 \\ 4! &= 24 \\ 5! &= 120 \\ 6! &= 720 \\ 7! &= 5040 \\ 8! &= 40320 \\ 9! &= 362880 \\ 10! &= 3628800 \end{aligned}$$

Factorial and permutations

For k many options that need to be ordered, there are k options for the first choice, $k-1$ for the second choice, and so on until the last (k-th position) is the only option remaining, so

there is only one choice for this one. As a result, there are $k \times (k-1) \times (k-2) \times \cdots \times 2 \times 1 = k!$ ways to order k options.

P12.3 The Python setting

To facilitate this project, some functions are provided in the `PreferencePermutations.py` file. It is not necessary to read and understand the code of these functions to be able to complete this project. The only thing that must be done is import it:

```
from PreferencePermutations import *
```

Let's take a look at the functions thus imported (only the ones that might be used, the auxiliary functions will not be described).

P12.3.1 `int2char`

The `int2char` function allows to convert a number into the corresponding letter. That allows a preference to be a string even when there are more than 10 candidates. Admittedly, that means there is a limit in the number of candidates, but using uppercase latin, lowercase greek, and lowercase latin, that allows for up to 76 candidates, which is already a lot of permutations given that 76! is about[1] 10^{111}.

When the argument is 0 or lower, the letter produced is `?`, which we can use to denote that the election did not produce a winner (for example in case of ties). It also produces a `?` (along with a warning) when the argument is above 76. Below is a table of the letter corresponding to the argument.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
?	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
T	U	V	W	X	Y	Z	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν

40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
ξ	o	π	ρ	σ	τ	υ	φ	χ	ψ	ω	a	b	c	d	e	f	g	h	i

60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77
j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	?

This function can be used, for example, to generate the list of candidates knowing how many there are.

```
print(int2char(0))
print(int2char(13))
print(int2char(42))
```

[1] To be exact:
76! = 1885494701666050254987932260861146558230394535379329335672487982961844043495537923117729972224000000000000000000

```
print(int2char(174))
```

```
?
M
π
?

[int2char] Warning: Could not convert number 174 to character. '?' will be␣
↪used.
```

P12.3.2 genAllPreferences and genAllPreferencesFromList

These functions are used to generate a list of all possible permutations. The first one, `genAllPreferences` takes an integer k and returns the list of all permutations for k candidates `A`, `B`, `C`...

The `genAllPreferencesFromList` function takes a list of strings and will use it to create the list of permutations of these strings. So `genAllPreferencesFromList(['A','B','C'])` is the same as `genAllPreferences(3)`.

The intended use of `genAllPreferencesFromList` is to be able to generate all permutations for a subset of candidates. Other uses, such as using strings with more than one letter should be avoided in the context of this project.

```
print("genAllPreferences(3):",genAllPreferences(3))
print("genAllPreferences(4):",genAllPreferences(4))
print("genAllPreferencesFromList(['A','B','D']):
↪",genAllPreferencesFromList(['A','B','D'])) # Intended use
print("genAllPreferencesFromList(['Hello','Good bye','Hi!']):
↪",genAllPreferencesFromList(['Hello','Good bye','Hi!'])) # Not advised
```

```
genAllPreferences(3): ['ABC', 'ACB', 'BAC', 'CAB', 'BCA', 'CBA']
genAllPreferences(4): ['ABCD', 'ABDC', 'ACBD', 'ADBC', 'ACDB', 'ADCB',␣
↪'BACD', 'BADC', 'CABD', 'DABC', 'CADB', 'DACB', 'BCAD', 'BDAC', 'CBAD',␣
↪'DBAC', 'CDAB', 'DCAB', 'BCDA', 'BDCA', 'CBDA', 'DBCA', 'CDBA', 'DCBA']
genAllPreferencesFromList(['A','B','D']): ['ABD', 'ADB', 'BAD', 'DAB',␣
↪'BDA', 'DBA']
genAllPreferencesFromList(['Hello','Good bye','Hi!']): ['HelloGood byeHi!
↪', 'HelloHi!Good bye', 'Good byeHelloHi!', 'Hi!HelloGood bye', 'Good␣
↪byeHi!Hello', 'Hi!Good byeHello']
```

P12.3.3 genRandomPreferences

The `genRandomPreferences` function takes two arguments: n, the number of voters, and k, the number of candidates.

It runs a randomized election where each of the n voter selects a permutation. The result of the election is given as a **dictionary** where keys are the orderings and values are the number of people who have this particular preference relation.

See Lecture L7.3 for a refresher on dictionaries.

For example, the result of `genRandomPreferences(10000,4)` could be:

```
{'ABDC': 406, 'DABC': 414, 'CBDA': 431, 'BADC': 436, 'ABCD': 467, 'CABD': 419,
 'CBAD': 398, 'BDCA': 433, 'BCDA': 395, 'BDAC': 407, 'DCBA': 400, 'DACB': 447,
 'BACD': 412, 'ACDB': 455, 'DCAB': 447, 'CDAB': 392, 'ACBD': 401, 'CADB': 396,
 'DBAC': 438, 'ADBC': 389, 'ADCB': 385, 'CDBA': 427, 'DBCA': 425, 'BCAD': 380}
```

Meaning 406 people have the preference, A is better than B, which is better than C, which is better than D; and 414 prefer D to anything, then A, then B, then C; and so on for the other 22 orderings.

This function is intended to generate some results in order to test the voting system functions.

```
print(genRandomPreferences(50000,3))
```

```
{'CAB': 8260, 'BCA': 8408, 'CBA': 8346, 'ABC': 8283, 'ACB': 8328, 'BAC':␣
↪8375}
```

P12.3.4 print and prettyPrintPreferences

The result of a vote can simply be printed with the built-in print function; it is actually possible to copy-paste the result to store in a variable. But it might not be so easy to read when debugging, especially if there are a lot of permutations that got 0 votes. This is why the prettyPrintPreferences is provided. The optional argument printZero (False by default) allows to control whether permutations without any votes will be displayed. It also tries to align values in a nice way.

```
someVote = {'CBDA': 0, 'CADB': 0, 'ACDB': 404, 'DABC': 442, 'CDBA': 426,␣
↪'DCBA': 425, 'BCDA': 0, 'CDAB': 398, 'DBAC': 0, 'DBCA': 428, 'BDAC':␣
↪426, 'CBAD': 428, 'ACBD': 414, 'DACB': 434, 'BDCA': 0, 'BACD': 410,␣
↪'ABDC': 0, 'ADBC': 0, 'ABCD': 446, 'CABD': 467, 'BADC': 0, 'DCAB': 386,␣
↪'BCAD': 0, 'ADCB': 392}
print(someVote)
prettyPrintPreferences(someVote,printZero=True)
prettyPrintPreferences(someVote,printZero=False) # Equivalent to␣
↪prettyPrintPreferences(someVote)
```

```
{'CBDA': 0, 'DABC': 442, 'DCAB': 386, 'CDAB': 398, 'CABD': 467, 'DBAC': 0,␣
↪'ADBC': 0, 'DBCA': 428, 'CADB': 0, 'BDCA': 0, 'CDBA': 426, 'ABCD': 446,␣
↪'ABDC': 0, 'BCDA': 0, 'BADC': 0, 'ADCB': 392, 'DCBA': 425, 'BACD': 410,␣
↪'DACB': 434, 'CBAD': 428, 'BCAD': 0, 'BDAC': 426, 'ACDB': 404, 'ACBD':␣
↪414}
 {
     CBDA: 0 ;     DABC: 442 ;   DCAB: 386 ;   CDAB: 398 ;
     CABD: 467 ;   DBAC: 0 ;     ADBC: 0 ;     DBCA: 428 ;
     CADB: 0 ;     BDCA: 0 ;     CDBA: 426 ;   ABCD: 446 ;
     ABDC: 0 ;     BCDA: 0 ;     BADC: 0 ;     ADCB: 392 ;
     DCBA: 425 ;   BACD: 410 ;   DACB: 434 ;   CBAD: 428 ;
     BCAD: 0 ;     BDAC: 426 ;   ACDB: 404 ;   ACBD: 414 ;
 }
```

```
{
    DABC: 442 ;   DCAB: 386 ;   CDAB: 398 ;   CABD: 467 ;   DBCA: 428 ;
    CDBA: 426 ;   ABCD: 446 ;   ADCB: 392 ;   DCBA: 425 ;   BACD: 410 ;
    DACB: 434 ;   CBAD: 428 ;   BDAC: 426 ;   ACDB: 404 ;   ACBD: 414 ;
}
```

P12.3.5 Other vote results for testing

Randomized votes tend to be quite uniform (because the random vote is chosen uniformly), so it may not be suited to testing some aspects of the voting system.

As a result, a result of a vote can be created by hand for tests. But if only a couple of permutations are actually used, it can be tedious to write them all. Therefore it is easier to generate a dictionary with all permutations with 0 votes each then change only the relevant values in the dictionary:

```python
vote4tedious = {'DBCA': 0, 'CBAD': 0, 'DACB': 0, 'BDAC': 0, 'CDBA': 0,␣
 ↪'CBDA': 0, 'ACDB': 51, 'ACBD': 0, 'BACD': 0, 'ABCD': 42, 'DCAB': 0,␣
 ↪'BCDA': 0, 'DBAC': 0, 'CADB': 0, 'ABDC': 0, 'ADCB': 0, 'BADC': 0,␣
 ↪'DCBA': 0, 'CDAB': 0, 'CABD': 0, 'BDCA': 107, 'DABC': 0, 'BCAD': 0,␣
 ↪'ADBC': 0}

vote4 = {p: 0 for p in genAllPreferences(4)} # Creates a dictionary with␣
 ↪keys taken in all permutations of 4 candidates, with value 0 for each
vote4['ABCD'] = 42
vote4['BDCA'] = 107
vote4['ACDB'] = 51

print("vote4tedious:")
prettyPrintPreferences(vote4tedious)
print("vote4:")
prettyPrintPreferences(vote4)
```

```
vote4tedious:
  {
       BDCA: 107 ;     ABCD: 42 ;      ACDB: 51 ;
  }
vote4:
  {
       BDCA: 107 ;     ABCD: 42 ;      ACDB: 51 ;
  }
```

P12.4 Voting systems

P12.4.1 Preferred ordering

In this voting system, the permutation with the most votes wins and the first choice of this permutation is chosen. This voting system is not really used anywhere because it is not fair

at all by any measure. It is however a very simple system so it is easy to program; and more importantly the ideas behind the program will be reused throughout the project.

Let's be more precise in what is expected:

- The function must take in a dictionary where the keys are the permutations and the value is the number of votes it obtained.
- It must return the winning candidate (one letter)
- In case of equality between two or more permutations with the highest number of votes, return `?` (using `int2char(0)`)
- Optionally: add an optional argument `verbose` initially `False` that switches the printing of debugging information.

The problem can be broken down in two phases. The function needs to: 1. Find the permutation with the most number of votes, if there is one 2. Take the first letter of this permutation

To find the "winning permutation", the following idea can be used:

- Keep in a variable the *champion permutation*, and in another variable the votes it obtained.
- Initially, there is "no champion"; that must be coded in some way, for example the empty string.
- For each permutation, compare the votes obtained by this permutation to the votes obtained by the champion.
- If the permutation obtained strictly more than the champion, then it becomes champion (and the votes obtained by the champion are also updated).
- If the permutation obtained as many votes as the champion, there is equality, so the current champion is "no champion"; the number of votes of the champion remains the same, however.
- If the permutation obtained fewer votes than the champion, nothing happens.

This idea can be turned into code:

```
def bestOrdering(votes,verbose=False):
    winner = ''                     # Initialize the champion to "no␣
↪champion"
    winCount = -1                   # And the number of votes the␣
↪champion has
    for p in votes:
        if votes[p] > winCount:     # We have a new champion
            winner = p              # Update champion
            winCount = votes[p]     # Update number of votes for champion
        elif votes[p] == winCount:  # Equality: there is no champion␣
↪anymore
            winner = ''
    if verbose:                     # Printing more info (if needed)
        if winner == '':
            print("Equality, no winner")
        else:
            print(winner,"is the preferred ordering␣
↪with",winCount,"votes")
```

```
    if winner == '':
        winner = int2char(0)        # No winner, return '?'
    else:
        winner = winner[0]          # First letter of the wining␣
↪permutation wins the election
    return winner
```

The function can be tested on custom and random voting results.

```
rndVotes = genRandomPreferences(50000,4)
customNoEqual = {p:0 for p in genAllPreferences(3)}
customNoEqual['ABC'] = 429
customNoEqual['CAB'] = 574
customNoEqual['BCA'] = 429
customNoEqual['BAC'] = 314
customNoEqual['CBA'] = 73
customChampEqual = {p:0 for p in genAllPreferences(3)}
customChampEqual['ABC'] = 712
customChampEqual['CAB'] = 324
customChampEqual['BCA'] = 214
customChampEqual['BAC'] = 712
customChampEqual['CBA'] = 67
prettyPrintPreferences(rndVotes)
print(bestOrdering(rndVotes,verbose=True))
print("---------------------------------------------------------")
prettyPrintPreferences(customNoEqual)
print(bestOrdering(customNoEqual,verbose=True))
print("---------------------------------------------------------")
prettyPrintPreferences(customChampEqual)
print(bestOrdering(customChampEqual,verbose=True))
```

```
   {
        CBDA: 2080 ;    DABC: 2049 ;    DCAB: 2051 ;    CDAB: 2120 ;
        CABD: 2081 ;    DBAC: 2047 ;    ADBC: 2124 ;    DBCA: 2130 ;
        ADCB: 2094 ;    BDCA: 2113 ;    CDBA: 2107 ;    ABCD: 2075 ;
        ABDC: 2071 ;    BCDA: 2104 ;    BADC: 2140 ;    CADB: 2022 ;
        DACB: 2107 ;    BACD: 2045 ;    DCBA: 2055 ;    CBAD: 2105 ;
        BCAD: 2095 ;    BDAC: 2124 ;    ACDB: 2020 ;    ACBD: 2041 ;
   }
BADC is the preferred ordering with 2140 votes
B
---------------------------------------------------------
   {
        CAB: 574 ;      BCA: 429 ;      CBA: 73 ;       ABC: 429 ;      ␣
↪BAC: 314
;
   }
CAB is the preferred ordering with 574 votes
```

```
C
-----------------------------------------------------------
   {
        CAB: 324 ;      BCA: 214 ;      CBA: 67 ;       ABC: 712 ;      ␣
 ↪BAC: 712
;
   }
Equality, no winner
?
```

P12.4.2 First past the post

In this system, the candidate which is placed as first preference by most voters is elected. This system is used in chamber elections in the US, Great-Britain, Canada, India, Pakistan, and many other countries. In most US states, the electoral college (that elects the President) is also chosen this way.

More precisely:

- The function must take in a dictionary where the keys are the permutations and the value is the number of votes it obtained.
- It must return the winning candidate (one letter)
- In case of equality between two or more candidates with the highest number of votes, return `?` (using `int2char(0)`)
- Optionally: add an optional argument `verbose` initially `False` that switches the printing of debugging information.

The problem can be broken down in two phases. The function needs to: 1. Tally the votes for each candidate into a dictionary where keys are the candidates. 2. Find the candidates with the most number of votes, if there is one.

Before discussing the procedure for the first point, let's remark that the second point is very similar to what was just performed for the **Preferred ordering**, except the dictionary used will contain votes for a candidate instead of a permutation.

Addressing the first point, the tallying of the votes: - First, create a dictionary with candidates as keys, and value 0 for each of them - Note that the number of candidates can be found by the length of the keys of the dictionary of votes - Then for each permutation, add the number of votes it obtained to the tally of the first letter, *i.e.* the preferred candidate.

Exercise

1. Complete the code below to produce the winner of first past the post election:

```python
def oneRound(votes,verbose=False):
    k = len(list(votes.keys())[0])          # Length of one preference␣
 ↪string.
    candidates = []                         # List of candidates
    for c in range(1,k+1):                  # Filled based on their number
        candidates.append(int2char(c))
    firstChoice = {c:0 for c in candidates} # Initialize count of how␣
 ↪many voters placed each candidate as first choice
```

```
    for p in votes.keys():
        firstChoice[p[0]] += votes[p]       # The first letter of the␣
↪permutation is the preferred candidate
    if verbose:                             # Printing the tally (if␣
↪needed)
        print("Votes obtained by each candidate (as first choice):")
        for c in candidates:
            print("\t",c,": ",firstChoice[c],sep="")
    ### Now that the votes have been tallied, code the part that finds␣
↪the winner of the election
```

2. Test this function on custom and randomized votes. Note that for a proper testing, some *weird* cases, for example cases where there is equality, should be tried.

```
#
# Code here
#
```

```
   {
        CBDA: 2026 ;    DABC: 2099 ;    DCAB: 2099 ;    CDAB: 2036 ;
        CABD: 2158 ;    DBAC: 2009 ;    ADBC: 2072 ;    DBCA: 2039 ;
        ADCB: 2061 ;    BDCA: 2048 ;    CDBA: 2153 ;    ABCD: 2081 ;
        ABDC: 2069 ;    BCDA: 2096 ;    BADC: 2043 ;    CADB: 2134 ;
        DACB: 2093 ;    BACD: 2158 ;    DCBA: 2091 ;    CBAD: 2088 ;
        BCAD: 2102 ;    BDAC: 2061 ;    ACDB: 2143 ;    ACBD: 2041 ;
   }
Votes obtained by each candidate (as first choice):
        A: 12467
        B: 12508
        C: 12595
        D: 12430
C wins with 12595 votes
C
---------------------------------------------------------
   {
        CAB: 574 ;      BCA: 429 ;      CBA: 73 ;       ABC: 429 ;      ␣
 ↪BAC: 314
;
   }
Votes obtained by each candidate (as first choice):
        A: 429
        B: 743
        C: 647
B wins with 743 votes
B
---------------------------------------------------------
   {
        CAB: 324 ;      BCA: 214 ;      CBA: 67 ;       ABC: 712 ;      ␣
 ↪BAC: 498
```

```
;
   }
Votes obtained by each candidate (as first choice):
        A: 712
        B: 712
        C: 391
Equality, no winner
?
```

P12.4.3 The Borda count method

This method was invented by Jean-Charles de Borda in the 18th century. Some modified versions of it are used to elect Members of the Parliament of Nauru and the President of the Kiribati.

It relies on the following principle: for each vote of a permutation with k candidates, the first choice gets k points, the second one gets $k-1$, and so on until the last choice gets 1 point. The candidate with the most points wins the election.

For example, if `ACB` has 37 votes, `BCA` gets 12 votes, and `CBA` gets 23 votes (and 0 votes for other permutations), the tally is as follows: - `A` gets $37 \times 3 = 111$ points from the 37 `ACB` permutations, $12 \times 1 = 12$ points from the 12 `BCA` permutations, and $23 \times 1 = 23$ points from the 23 `CBA` permutations. That is a total of 146 points. - `B` gets $37 \times 1 = 37$ points from the 37 `ACB` permutations, $12 \times 3 = 36$ points from the 12 `BCA` permutations, and $23 \times 2 = 46$ points from the 23 `CBA` permutations. That is a total of 119 points. - `C` gets $37 \times 2 = 74$ points from the 37 `ACB` permutations, $12 \times 2 = 24$ points from the 12 `BCA` permutations, and $23 \times 3 = 69$ points from the 23 `CBA` permutations. That is a total of 167 points.

So `C` is the winner here.

Exercise

Write the function `borda` that takes the vote result (a dictionary with the number of votes for each permutation) and an optional `verbose` argument defaulting to `False` (for debug purposes) and returns the winner according to the Borda count method. Remark that, similarly to the previous case, the function works in two phases: tally the points, then find the winner. In case of equality of points, return `?`.

```
#
# Code here
#
```

P12.4.4 The Condorcet method

The Condorcet method, named after its inventor Nicolas de Condorcet, is based on the following principle: the winner should be the candidate that is preferred to others in a duel. That criterion is not met by first past the post or two-round elections that were mostly used at the time, motivating Condorcet to invent this method that ensures it by construction. It can however very often produce ties, so lots of variants have been designed to split these situations, collectively referred to as "Condorcet methods". Only work on the simplest form is considered here.

Since the Condorcet method relies on *duels*, the code needs to be able to determine the winner of a duel from the results of a k-candidate vote. The principle is easy: candidate c_1 is preferred to candidate c_2 if in more votes c_1 is before c_2.

For example, in permutations `ABC`, `ACB`, and `CAB` candidate `A` is preferred to candidate `B`. So votes for all these permutations count as votes for `A`; votes for the oher three permutations (`BAC`, `BCA`, `CBA`) would count as votes for `B`.

More precisely, for a given permutation, if the letter c_1 appears first, the votes are tallied to c_1. If the letter c_2 appears first, the votes are tallied to c_2. The winner is the one with more votes at the end.

Exercise

Code the `duel` function that takes the vote result (a dictionary with the number of votes for each permutation), candidates `c1` and `c2` and an optional `verbose` argument defaulting to `False` (for debug purposes) and returns the winner of the duel between `c1` and `c2` (or `?` in case of equality).

```
#
# Code here
#
```

This `duel` function can be used to determine the winner according to Condorcet's method. Each candidate need to be confronted with the others in duels, and the function counts how many of these are won by each candidate (if there is equality in a duel, no one wins). At the end, the candidate that wins the most duels is the winner.

Exercise

1. Code the `condorcet` function that takes the vote result (a dictionary with the number of votes for each permutation), and an optional `verbose` argument defaulting to `False` (for debug purposes) and returns the winner of the vote according to the Condorcet method (or `?` in case of equality).
2. Test the function on the two examples below: `noCondorcetWin` produces an equality while with `condorcetBwins` `B` is the winner.

```
noCondorcetWin = {'CBDA': 6, 'BDAC': 3, 'BCAD': 7, 'CBAD': 2, 'ACDB': 5,
 ↪'CDBA': 5, 'DACB': 4, 'CDAB': 3, 'BACD': 9, 'CABD': 6, 'ABDC': 2,
 ↪'ADBC': 2, 'ACBD': 5, 'CADB': 6, 'BDCA': 6, 'DCAB': 1, 'ABCD': 1,
 ↪'DABC': 6, 'DCBA': 3, 'BCDA': 6, 'DBCA': 0, 'BADC': 4, 'DBAC': 4,
 ↪'ADCB': 4}
condorcetBwins = {'BDAC': 9, 'CDBA': 3, 'BCDA': 7, 'ADBC': 5, 'DBAC': 2,
 ↪'CABD': 4, 'ABDC': 6, 'BACD': 0, 'BCAD': 4, 'DABC': 4, 'DBCA': 8,
 ↪'ACDB': 3, 'ADCB': 1, 'ABCD': 2, 'CBDA': 3, 'DACB': 6, 'CBAD': 3,
 ↪'BDCA': 3, 'CDAB': 4, 'ACBD': 6, 'BADC': 8, 'DCAB': 6, 'CADB': 1,
 ↪'DCBA': 6}
```

```
#
# Code here
#
```

P12.4.5 Two round voting

This system is an extension of first-past the post, in which if a candidate did not gather strictly more than half the votes (*absolute majority*), a second round is organized with the two higher-ranking candidates.

In order to create a function that determines the winner in this system, let's compare it with some systems we have seen before.

First thing to remark is that finding the two preferred candidates is similar to finding the candidates with the most votes, except that variables to store the runner-up and its number of votes also need to be kept. There are quite many cases that need to be considered when going through the candidates to determine the two that will take part in the second round.

Let's call C the challenger, that is to say the candidate being considered when going through the list of candidates. Let F be the current front-runner and R be the current runner-up. Let $v(x)$ be the number of votes (as preferred candidate) candidate x obtained. The situation might depend on how F and R compare before the challenge: if $v(F) = v(R)$, meaning equality, things will be different.

- Assume $v(F) > v(R)$ (front runner have more votes than runner-up)
- If $v(C) > v(F)$ (challenger is better than everyone seen so far), then F becomes R and C becomes F
- If $v(F) = v(C)$ (challenger equals front runner), then C becomes R
- If $v(F) > v(C) > v(R)$ (challenger is between front-runner and runner-up), then C becomes R
- If $v(F) > v(C) = v(R)$ (challenger equals runner-up), then there is equality for runner up and R becomes "nobody" (`?` or `int2char(0)` in the code)
- If $v(F) > v(R) > v(C)$ (challenger is lower than runner up), nothing changes
- Assume $v(F) = v(R)$ (equality between current two leaders)
- If $v(C) > v(F) = v(R)$ (challenger above the equal leaders), then C becomes F and "nobody" becomes the runner-up.
- If $v(C) = v(F) = v(R)$ (challenger same as equal leaders), then "nobody" becomes F and R (triple equality)
- If $v(F) = v(R) > v(C)$ (challenger is lower than runner up) nothing changes

While these cases can be grouped in a bunch of different ways in the code, in the end all must have been considered and treated accordingly.

Then the second round is a duel, the corresponding function written for the Condorcet method can be used (*i.e.* called) .

P12.4.6 Exercise

1. Code the `twoRounds` function that takes the vote result (a dictionary with the number of votes for each permutation), and an optional `verbose` argument defaulting to `False` (for debug purposes) and returns the winner of the vote according to the two-round voting method (or `?` in case of equality).

```
#
# Code here
#
```

2. Test your function on randomized and custom tests. The following cases should be part of the tests:
    - a candidate wins in the first round
    - there is equality in the second round
    - there is equality between two (but not three) leaders in the first round
    - there is equality between three leaders in the first round
    - there is a front runner but two equal runner-ups in the first round
    - there is no equality ("normal" case)

```
#
# Code here
#
```

P12.4.7 Elimination voting

Officially known as *instant runoff*, this voting system works by eliminating the candidate with the least amount of votes as a first choice until there remains only a single candidate. Of course, the votes that placed an eliminated candidate first do count in the subsequent rounds with their second choice as their "new first choice".

For example, if there are 4 candidates `A`, `B`, `C`, `D` and votes as follows:

- `BCDA` gets 1241 votes
- `ABCD` gets 1074 votes
- `CDAB` gets 914 votes
- `CADB` gets 789 votes
- `ACBD` gets 702 votes
- `DACB` gets 598 votes
- `ADCB` gets 349 votes

In the first round, `A` is ranked first by 2125 voters, `B` by 1241 voters, `C` by 1703 voters, and `D` by 598 voters. So `D` is eliminated. The above preferences can be rewritten without `D`:

- `BCA` gets 1241 votes
- `ABC` gets 1074 votes
- `CAB` gets 914 votes
- `CAB` gets 789 votes
- `ACB` gets 702 votes
- `ACB` gets 598 votes
- `ACB` gets 349 votes

Of course some preferences that differed only by the position of `D` are now the same and should be summed:

- `BCA` gets 1241 votes
- `ABC` gets 1074 votes
- `CAB` gets 914+789=1703 votes
- `ACB` gets 702+598+349=1649 votes

So in the second round, `A` is ranked first by 2723 voters, `B` by 1241 voters, `C` by 1703 voters. So `B` is eliminated, which allows to rewrite the preferences as:

- `CA` gets 1241 votes
- `AC` gets 1074 votes
- `CA` gets 1703 votes
- `AC` gets 1649 votes

Gathering the preferences that appear twice:

- `CA` gets 1241+1703=2944 votes
- `AC` gets 1074+1649=2723 votes

So in the third round `A` is eliminated and `C` wins. (This example is provided as `exElim` below, in order to test the `elimination` function.)

The tallying of votes as first choice is similar to what was performed before. The main changes here are that:

- The loser is being sought out, instead of the winner.
  - The number of votes for the loser should be initialized to a number higher than any candidate obtained. For example, the total number of voters+1.
- The list of candidates and the votes themselves will change when eliminating a candidate.
  - To remove an element from a list, use the `del` operation on the element in the list; but for that the index of the element must be provided.
  - To find the index of an element in the list, use the `index` method. For example

```
l = ["Hello","Bye","Hi"]
l.index("Hi") # returns 2
del l[2]      # deletes "Hi"
```

  - To delete a candidate, which is a single character, from a permutation, which is a string, the `replace` method can be used. This method takes two strings and replaces every occurrence of the first string by the second one:

```
s = "Hello world!"
s1 = s.replace('o','0')  # s1 is "Hell0 w0rld!" (every o turned to 0)
s2 = s.replace('l','')   # s2 is "Heo word!" (every l removed)
s3 = s.replace('ll','X') # s3 is "HeXo world!"
                  (every double l turned to X, single l not changed)
```

  - To generate the list of possible permutations use the provided function `genAllPreferencesFromList` from the list of candidates still in play.
  - A new dictionary of votes gathers the votes after elimination of the loser. This will become the new votes.

Exercise

Code the `elimination` function that takes the vote result (a dictionary with the number of votes for each permutation), and an optional `verbose` argument defaulting to `False` (for debug purposes) and returns the winner of the vote according to the elimination voting method (or `?` in case of equality).

```
#
# Code here
#
```

```
exElim = {p: 0 for p in genAllPreferences(4)}
exElim['BCDA'] = 1241
exElim['ABCD'] = 1074
exElim['CDAB'] = 914
exElim['CADB'] = 789
exElim['ACBD'] = 702
exElim['DACB'] = 598
exElim['ADCB'] = 349
prettyPrintPreferences(exElim)
# print(elimination(exElim,verbose=True)) # Uncomment when elimination␣
↪has been defined
```

P12.4.8 Comparison of the voting methods (Exercise)

Execute all the voting methods we have coded on the `vote5` vote results below. What can you conclude?

```
vote5 = {p: 0 for p in genAllPreferences(5)}
vote5['ADECB'] = 3273
vote5['BEDCA'] = 2182
vote5['CBEDA'] = 1818
vote5['DCEBA'] = 1636
vote5['EBDCA'] = 727
vote5['ECDBA'] = 364
```

Project 13
Protein Translation

Project contents

P13.1 The protein translation process **211**
P13.2 Step 1 - DNA to mRNA Transcription **212**
P13.2.1 Reading the file (Exercise) 212
P13.2.2 DNA to mRNA transcription (Exercise) 213
P13.3 Step 2 - mRNA to Protein Translation **213**
P13.3.1 A codon dictionary (Exercise) 214
P13.3.2 Translating using the codon dictionary (Exercise) 214
P13.4 Step 3: A `main` function to connect the processes (Exercise) **215**

P13.1 The protein translation process

How do living organisms produce proteins which are structures responsible from all the major functions of a cell? In this project, you will be learning about the entire process of protein translation and creating different protein structures of a strand of E.coli bacteria. For extensive information, the following article is suggested as a good reading:

How does the cell convert DNA into working proteins? by Clancy et al.

The protein translation process starts with DNA to mRNA transcription:

> “A DNA sequence has a double helix structure that looks like a staircase consisting of base pairs. There are four types of bases (nucleotides) in a DNA molecule: Adenine (A), Thymine (T), Guanine (G) and Cytosine (C). The bases are paired on a sugar-phosphate backbone structure. Adenine base is always paired with Thymine and Guanine base is always paired with Cytosine.”
>
> (From notes of E. Yildirim
> *Designing Computational Biology Workflows with Perl - Part 2*)

mRNA molecules are created from a single strand of DNA, through base pair matching with one exception. Instead of Tyhmine, mRNA has a base called Urasil (U).

For example: If the DNA strand has the following bases ATGCCCGTTA, its corresponding mRNA is UACGGGCAAU. A is paired with U, T is paired with A, C is paired with G and G is paired with C.

After mRNA is transcribed, it is translated into codons which are triplets of bases. Each codon has a special meaning and corresponds to a specific aminoacid. Then, aminoacids are sequenced to create a protein.

How do we indicate the start and end of an aminoacid sequence? As it turns out, some codons are reserved to indicate this. For example, `AUG` codon indicates the start of the protein synthesis, while three other codons indicate the end: `UAG`, `UGA`, `UAA`.

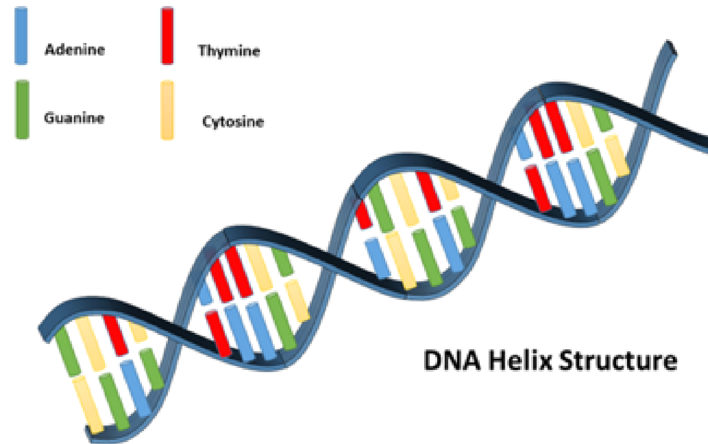


Figure P13.1: DNA double helix structure (From notes of E. Yildirim *Designing Computational Biology Workflows with Perl - Part 2*)

P13.2 Step 1 - DNA to mRNA Transcription

In the project folder, there are two files that need to be used for the protein translation of E.coli bacteria. The first file `ecoli.fa` is a FASTA file which contains the DNA sequence data. Here is an excerpt from the file:

```
>Chromosome dna_rm:chromosome chromosome:ASM584v2:Chromosome:1:4641652:1 REF
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTC
TGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGG
TCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTAC
ACAACATCCATGAAACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGT
AACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGACAGTGCGGG
CTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGT
ACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTTGCCGATATTCTGGAAAGCAATGCC
AGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTG
GCGATGATTGAAAAAACCATTAGCGGCCAGGATGCTTTACCCAATATCAGCGATGCCGAA
CGTATTTTTGCCGAACTTTTGACGGGACTCGCCGCCGCCCAGCCGGGGTTCCCGCTGGCG
CAATTGAAAACTTTCGTCGATCAGGAATTTGCCCAAATAAAACATGTCCTGCATGGCATT
AGTTTGTTGGGGCAGTGCCCGGATAGCATCAACGCTGCGCTGATTTGCCGTGGCGAGAAA
ATGTCGATCGCCATTATGGCCGGCGTATTAGAAGCGCGCGGTCACAACGTTACTGTTATC
GATCCGGTCGAAAAACTGCTGGCAGTGGGGCATTACCTCGAATCTACCGTCGATATTGCT
GAGTCCACCCGCCGTATTGCGGCAAGCCGCATTCCGGCTGATCACATGGTGCTGATGGCA
GGTTTCACCGCCGGTAATGAAAAAGGCGAACTGGTGGTGCTTGGACGCAACGGTTCCGAC
TACTCTGCTGCGGTGCTGGCTGCCTGTTTACGCGCCGATTGTTGCGAGATTTGGACGGAC
GTTGACGGGGTCTATACCTGCGACCCGCGTCAGGTGCCCGATGCGAGGTTGTTGAAGTCG
ATGTCCTACCAGGAAGCGATGGAGCTTTCCTACTTCGGCGCTAAAGTTCTTCACCCCCGC
```

P13.2.1 Reading the file (Exercise)

The first line of the file is information about the file. The rest is the actual DNA sequence data. Write a function `read_file` that reads the entire file into a string variable and returns it:

```
def read_file(file_path):    # file_path holds the path of the ecoli.fa␣
↪file
    # To exclude the first line, read the file line by line
    fp = open(file_path,"r") # Opens a file to read and creates a file␣
↪handle fp
    line = fp.readline()     # Reads a line from the file and stores it␣
↪in line variable
    # Create a loop and read the file line by line until you read an␣
↪empty string
    # in each iteration clean the string from any whitespace characters␣
↪at the beginning and
    # at the end. line.strip() will return the stripped string
    # then concatenate into dna_string variable.
    #
    # Complete the code here
    #
    return dna_string
```

P13.2.2 DNA to mRNA transcription (Exercise)

Write a function `transcribe(dna_string)` that creates the mRNA string from the DNA string. Each base in `dna_string` must be matched to its corresponding mRNA base. There might be strange characters in the DNA string other than A, T, C, G. They should be ignored: A $\rightarrow$ U, T $\rightarrow$ A, G $\rightarrow$ C, C $\rightarrow$ G matchings are the only valid ones.

```
#
# Code here
#
```

P13.3 Step 2 - mRNA to Protein Translation

The second file in the project folder is a CSV file named `codon_table.csv` which contains the `codon` list. Here is an excerpt from the file:

Codon	AA.Abv	AA.Code	AA.Name
UUU	Phe	F	Phenylalanine
UUC	Phe	F	Phenylalanine
UUA	Leu	L	Leucine
UUG	Leu	L	Leucine
CUU	Leu	L	Leucine

In the above table, `AA.Abv` represents the abbreviation of the aminoacid, `AA.Code` represents the code for the aminoacid and `AA.Name` represents the actual name of the aminoacid. There are 64 codons in the file. One aminoacid can be represented with multiple codons, they all create the same aminoacid. For example, both `UUU` and `UUC` codons are translated as `phenylalanine`.

P13.3.1 A codon dictionary (Exercise)

Write a function `read_csv(file_path)` that accepts the path of the codon list file and creates and returns a dictionary where key is codon and value is its corresponding `AA.Code`. The `csv` module must be imported before the `csv.reader` function can be used.

```
import csv
def read_csv(file_path):
    with open(file_path, 'r', newline='') as file_handle:
        reader = csv.reader(file_handle, delimiter = ',')
        # Everything from this point on must be indented under 'with'
        codon_dict = {}
        counter = 0 # Use this for the first row which includes the␣
↪headers of the columns
        # Create a for loop to read the reader list that consists of row␣
↪tuples
        for row in reader:
            # row[0] represents Codon, row[2] represents AA.Code
            #
            # Complete the code here
            #
            pass # Delete this when your code has been written
    return codon_dict # The codon dictionary
```

P13.3.2 Translating using the codon dictionary (Exercise)

Write a function `translate` which accepts the mRNA string and codon dictionary as parameters and creates a dictionary of proteins. The key of the dictionary could be a number starting from 0,1,2, ... and continues to increase as new protein aminoacid strings are added. The value part of the dictionary must hold the `AA.Code` string of the protein. The function should return the dictionary as a result.

Each protein's aminoacid sequence starts with M (Methionine) which is the starting aminoacid and ends with a Stop aminoacid. So the function should:

- look for mRNA sequences that starts with AUG codon;
- detect the end (UAG, UGA, or UAA codon);
- in between, identify the corresponding aminoacids for the codons to construct the protein;
- save the protein in the dictionary.

```
def translate(mrna_string, codon_dict):
    # Starting codon : AUG
    # Ending codon : UAG UAA UGA
    # Each Aminoacid sequence start with AUG codon that translate to␣
↪Methionine(M) aminoacid
    # The sequence might end with one of the stop codons UAG, UAA, or UGA.
    #
    # Complete the code here
    #
```

```
    return dct_proteins
```

P13.4 Step 3: A `main` function to connect the processes (Exercise)

Write the `main` function that ties all these steps together:

1. Read the `ecoli.fa` file
2. Call the `transcribe` function
3. Read the `codon_list.csv` file
4. Call the `translate` function
5. Print the resulting dictionary

The first few lines of the output should look like this:

```
0 -> MDGTHLILKStop
1 -> MKLVISVSRVCLFLMSHVLStop
2 -> MVVVVVMVSIATPDCACPLCLFSGVDCHARKKKLVSIAPLLVRSQLQAAMStop
3 -> MGYSRYGLHKNGLKTALSGGGSAPRATALTFESSStop
4 -> MTIARPAFStop
5 -> MELRWQLStop
6 -> MRRRHDRRTNARLTTLStop
7 -> MDAGRSPRATLQQLQLQDGPSLPRKDEAAISRSGGVVMGVAGQGLGNGLIFMAFRSSWSMRVTTVGTTSAStop
8 -> MRStop
9 -> MSStop
10 -> MDLDFLPNDLGDRHCLADRStop
11 -> MSSLRQVMASPIASQTLTAATATATRRPRStop
12 -> MHRRHNGStop
....
```

```
#
# Code here
#
```