# Introduction to Python Programming for Biologists

**5-day Workshop**

**8-12 April 2019**

***Room 2005***

***Perold building***

***Stellenbosch***

**Bring your Own Notebook Computer**

Contents

[Contents 2](#_Toc3187567)

[Introduction 4](#_Toc3187568)

[Data structures 5](#_Toc3187569)

[Bytes 5](#_Toc3187570)

[Hexadecimal 6](#_Toc3187571)

[Big-endian and little-endian 7](#_Toc3187572)

[Memory 7](#_Toc3187573)

[A series of instructions 7](#_Toc3187574)

[Install Python 3.x 8](#_Toc3187575)

[Python is an interpreter 8](#_Toc3187576)

[Install PyCharm 9](#_Toc3187577)

[Launch PyCharm 9](#_Toc3187578)

[Comments in code 13](#_Toc3187579)

[Variables 13](#_Toc3187580)

[Operators 14](#_Toc3187581)

[Equality and assignment operators 15](#_Toc3187582)

[Bitwise and logical operators 16](#_Toc3187583)

[Data Types 17](#_Toc3187584)

[Strings 17](#_Toc3187585)

[Exercises Day 1 21](#_Toc3187586)

[Lists, tuples, sets and dictionaries 25](#_Toc3187587)

[Data Input and Output 30](#_Toc3187588)

[Input from the keyboard 30](#_Toc3187589)

[Reading data from a file 30](#_Toc3187590)

[Manipulating bits and bytes in Python 32](#_Toc3187591)

[Representation of Characters 33](#_Toc3187592)

[Writing data to a file 35](#_Toc3187593)

[Exercises Day 2 37](#_Toc3187594)

[Loops 41](#_Toc3187595)

[The for-loop 41](#_Toc3187596)

[Enumerate 42](#_Toc3187597)

[Nested loops 42](#_Toc3187598)

[The while-loop 43](#_Toc3187599)

[Conditionals 44](#_Toc3187600)

[If, elif, else 44](#_Toc3187601)

[Exercise Day 3 46](#_Toc3187602)

[Functions 50](#_Toc3187603)

[Simple Functions 50](#_Toc3187604)

[Variable Scope 51](#_Toc3187605)

[Recursive functions 53](#_Toc3187606)

[Classes 55](#_Toc3187607)

[Classes 55](#_Toc3187608)

[Inheritance 57](#_Toc3187609)

[Containers and Iterations 58](#_Toc3187610)

[Assignment Day 4 60](#_Toc3187611)

[Modules 63](#_Toc3187612)

[Importing modules 63](#_Toc3187613)

[Numpy 64](#_Toc3187614)

[Exceptions 68](#_Toc3187615)

[Debugging 71](#_Toc3187616)

[Breakpoints and conditional breakpoints 72](#_Toc3187617)

[Single stepping 73](#_Toc3187618)

[Assignment Day 5 74](#_Toc3187619)

[Appendix 80](#_Toc3187620)

[File formats 80](#_Toc3187621)

[fastA 80](#_Toc3187622)

[fastQ 81](#_Toc3187623)

[Generic Feature Format version 3 (GFF3) 82](#_Toc3187624)

## Introduction

This **workshop** introduces you to **Python 3** for **programming** in the **biological sciences**. This is presented over **5 days**, and includes exercise sections to **test your mastering of concepts** encountered on each day. By the end of day 4, you will have developed to a point where you can code a modest program, and your practical assignment will be to code a program that can read a multi sequence fastA format file, and calculate the AT% of each sequence in the file.

On day 5 you are given a slightly more challenging assignment where you will read the sequences of the 17 *Saccharomyces cerevisiae* chromosomes (including the mitochondrial chromosome), using your fastA program from day 4, but will then also read information from a GFF format file of gene features that specifies the location of all genes in the *S. cerevisiae* genome. You will use this information to select all yeast gene sequences and calculate the frequency of all possible triplet codon sequences in the gene encoding sequences.

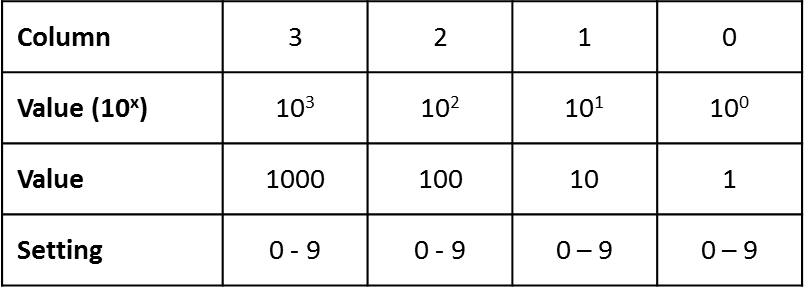
At this point, this may sound incredibly complex, but **it is not!** This course will provide you will all the **coding knowledge** that you **will need** to code the programs for the day 4 and day 5 exercises. In addition, this course will provide you with a **solid programming foundation** in Python 3 to write many programs that you can use in your **own scientific research**, and from which you can grow and gain experience to become a **bioinformatician** or **scientific programmer**. This is the era of **Big Data**, and it is **essential** in the **biological sciences** to be able **to code**. Once you are **comfortable** with using Python 3, it will be relatively **easy** to also **learn other languages** such as **Java** or **C/C++**.

This course is carefully structured to introduce concepts in an order required to **understand successive ideas** as we **build upon levels of sophistication**, and to provide **insight** into **basic concepts** that will enhance your understanding of programming as we go along. **Enjoy the course!** Remember, with a computer, **tinkering** and **trying things** are **encouraged!**

## Data structures

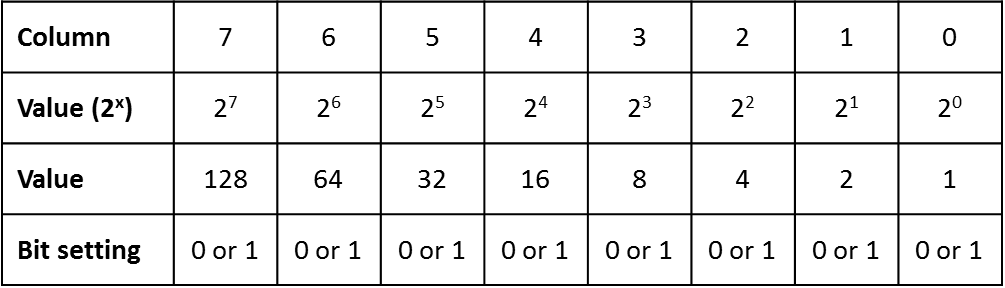
### Bytes

Humans count in units of 10, probably because we have 10 fingers.



***Fig. 1. Humans count in a base 10 system.*** *When column 0 of value 1 contains 9 dots it is “full”. If you add a tenth dot, you need to add one dot to the column 1 of value 10, and remove the 9 dots from column 0.*

**Computers** represent **numbers** using small hardware switches called **transistors**. The transistors can only be **on or off**, it cannot be one of 10 (0 to 9) different states. For this reason, computers “count” in **base 2**.



***Fig. 2. Use the base 2 system to count using computer transistors.*** *Like the base 10 scheme, where columns represent 1 (100), 10 (101), 100 (102), the base 2 (or binary) scheme represents 1 (20), 2 (21), 4 (22), 8 (24), etc. The value of a bit (“binary information unit”; 0 or 1) depends on its position. In column 0 a bit represents 0 or 1, in column 1 it represents 0 or 2, in column 3, 0 or 4, etc.*

***Note***

The value of a **set bit** (bit that is “on”) in a column is one greater than the sum of the maximum values of all the columns to the right. This is, of course, identical to the base 10 system (see Fig. 1) where, say, 100 is 1 greater that the maximum setting (99) for column 1 (10s) and column 0 (1s).

The 8 columns (0 - 7), representing **8 bits** is known as **1 byte**. The terms for other groups of bits are shown in Table 1.

***Table 1. The largest integer number that can be represented by different numbers of bits, and the term for each collection.***

|  |  |  |
| --- | --- | --- |
| **Number of bits** | **Largest number represented** | **Term** |
| 4 | 15 | nibble |
| 8 | 255 | byte |
| 16 | 65,535 | word |
| - | 1,024 | kilobyte (KB) |
| - | 1,048,576 | megabyte (MB) |
| - | 1,073,741,824 | terabyte (TB) |
| 64 | 1.845×1019 | - |

### Hexadecimal

An additional counting system that is often encountered in computing is **base 16**, or the **hexadecimal** system. The individual “number” in a group of 16 is represented by 0-9 continuing with the letters A, B, C, D, E and F, thus 0-F.

***Note***

You need 4 bits to represent 16 different values (see Fig. 2). Thus, **each byte of 8 bits** can be represented by **2 hexadecimal numbers**. Each hexadecimal number represents a **nibble**. Examples of hexadecimal representations are shown in Table 2. It is useful to know that if all 8 bits are set in a byte, the number represents decimal 255, which is equivalent to hexadecimal FF. If all bits in a 64 bit number is set, the hexadecimal representation would be FFFF FFFF FFFF FFFF. It is convention to write **each 16 bit value** as a **hexadecimal block** for clarity.

***Table 2. The decimal, binary and hexadecimal representation of some numbers.***

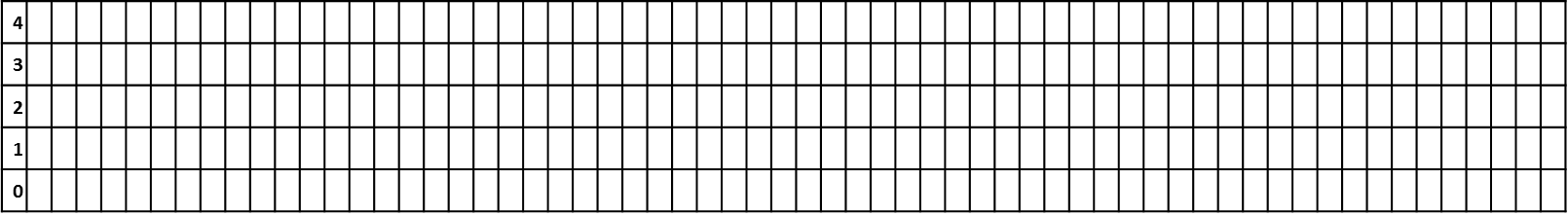
|  |  |  |
| --- | --- | --- |
| **Decimal value** | **Binary value** | **Hexadecimal value** |
| 15 | 00001111 | 0F |
| 32 | 00100000‬‬‬‬‬‬‬‬‬‬‬‬ | 20 |
| 204 | 11001100‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬ | CC |
| 255 | 11111111 | FF |
| 65,535 | 1111111111111111‬‬‬‬‬‬‬‬‬‬‬‬ | FFFF |

### Big-endian and little-endian

When you have a number represented by more than one byte, say 8CDF, the order of the bytes becomes important. For instance, hexadecimal 8CDF is 36063 in decimal, but the reverse DF8C is 57228. You therefore must have a system that defines whether the higher value byte (8C) is written first or last. There are two possibilities, called **big-endian** and **little-endian**, based on the Lilliputians in Gulliver’s Travels who argued whether an egg should be broken at its “little end” or “big end”. The big-endian system defines that the high value byte comes first. You would typically not have to worry about this when you program -- the operating system takes care of it. But if you read and write multi-byte values or floats to memory, you must know whether the memory arrangement is big- or small-endian.

## Memory

Memory is typically a range of locations, where each location has a unique address (Fig. 3). Each memory locations can typically store 64 bits of information. The largest 64 bit number that can be stored is 9,223,372,036,854,775,807 although there is no upper limit to an integer in Python.



***Fig. 3.*** *Memory is composed of a number of 64 bit locations, where each location has a unique address, here indicated as 0, 1, etc.*

## A series of instructions

A computer program is simply a list of instructions that is executed by a computer. This is similar to explaining the route from the town square parking lot to your favorite restaurant:

1. At the parking lot pedestrian exit, cross the street to the opposite sidewalk.
2. Turn left.
3. Continue walking until you are at the traffic light.
4. If the light is green, cross to the opposite sidewalk.
5. Turn right.
6. Walk and count 35 steps.
7. Turn left.
8. You are now in front of your favorite restaurant.

Although this is a mundane set of instructions, it contains several features found in real computer programs. In step 3, you continue doing something *until* a condition is met: you are at a traffic light. In step 4, you do something *if* something is true: the lights is green. In step 6, you repeatedly execute an action: you take 35 steps. As you will see later in this practical, a computer program typically also repeats a series of instructions a number of times or until a condition is met, and will also execute a series of instructions only if a certain condition is True (or False).

At a very low level, instructions would be reading the contents of one or more memory locations, copying the contents to the CPU, doing something with those values, and storing the result back to another memory location. However, high level languages such as Python or Java or C++ allows you to write instructions that look more like the instructions describing the route to the restaurant, than operating on the content of memory locations. However, it is useful to know what happens “under the hood”, so that programming does not become detached from the functioning of the computer.

Python was originally developed by Guido Van Rossum, and released in February 1991. Its name is derived from the BBC comedy series “[Monty Python’s Flying Circus](https://en.wikipedia.org/wiki/Monty_Python's_Flying_Circus)”. One of the most famous Monty Python routines is the [Dead Parrot Sketch](https://www.youtube.com/watch?v=4vuW6tQ0218). Python is an interpreted, object-oriented programming language, incorporating modules, exceptions, dynamic typing, very high level dynamic data types, and classes.

There are many excellent sources of help on the Internet that you can turn to for assistance. Make use of these when you work through this practical. Bookmark the [official Python documentation](https://docs.python.org/3/reference/index.html) and the [Python tutorial](https://docs.python.org/3/tutorial/index.html) and refer to these for additional help. You can also refer to the following books:

* [Lubanovic, B. Introducing Python. O’Reilly Media Inc. 2014 ISBN-13: 978-1449359362](https://www.amazon.com/Introducing-Python-Modern-Computing-Packages/dp/1449359361/ref=sr_1_1?s=books&ie=UTF8&qid=1516353647&sr=1-1&keywords=Introducing+Python)
* [Ramalho, L. Fluent Python. O’Reilly Media Inc. 2015 ISBN-13: 978-1491946008](https://www.amazon.com/Fluent-Python-Concise-Effective-Programming/dp/1491946008/ref=sr_1_1?s=books&ie=UTF8&qid=1516353420&sr=1-1&keywords=Fluent+Python.)
* [Lutz, M. Learning Python. O’Reilly Media Inc. 2013 ISBN-13: 978-9351102014](https://www.amazon.com/Learning-Python-Powerful-Object-Oriented-Programming/dp/9351102017/ref=sr_1_1?s=books&ie=UTF8&qid=1516353457&sr=1-1&keywords=%E2%80%A2%09Lutz%2C+M.+Learning+Python.)

## Install Python 3.x

You can install the latest version of Python 3 on Windows, MacOS or Linux by downloading and installing the binaries from [www.python.org](http://www.python.org/). Full installation instructions are given on the web site. There are several documents, guides and forums accessible on the website where you can get advice if you struggle.

***Note***

Python 3.x is already installed on the Narga lab computers.

### Python is an interpreter

Computer programs can be executed in one of two ways: the program can be compiled, which means that a compiler processes the entire series of instructions, and generates a series of low level instructions that is executed by the CPU. An example of a compiled language is C and C++. The advantage of a compiled language is that the compiled code generally executes very fast, and this type of program is therefore well suited to computation-intensive tasks.

An interpreted program is taken one line of instruction at a time, translated to a low level code that is executed by the CPU. Interpreted languages typically execute slower, but is more convenient to learn, because the program can be executed by passing it to the interpreter, and does not need to go through a compilation step, that requires time.

Thus, for an interpreted language like Python you would write a series of (meaningful) instructions in a text file, and then provide this text file to the python interpreter to interpret and execute. For instance, if you program file was called “my\_program.txt”, you would execute it with the command:

>python my\_program.txt

However, writing programs in text files, and passing it to the python interpreter in command line is tedious. It is much easier to use an application development environment where you can write, test, debug and run your program without having to leave the development application. One example of such an application is PyCharm. We will be using PyCharm extensively in this practical.

***Note***

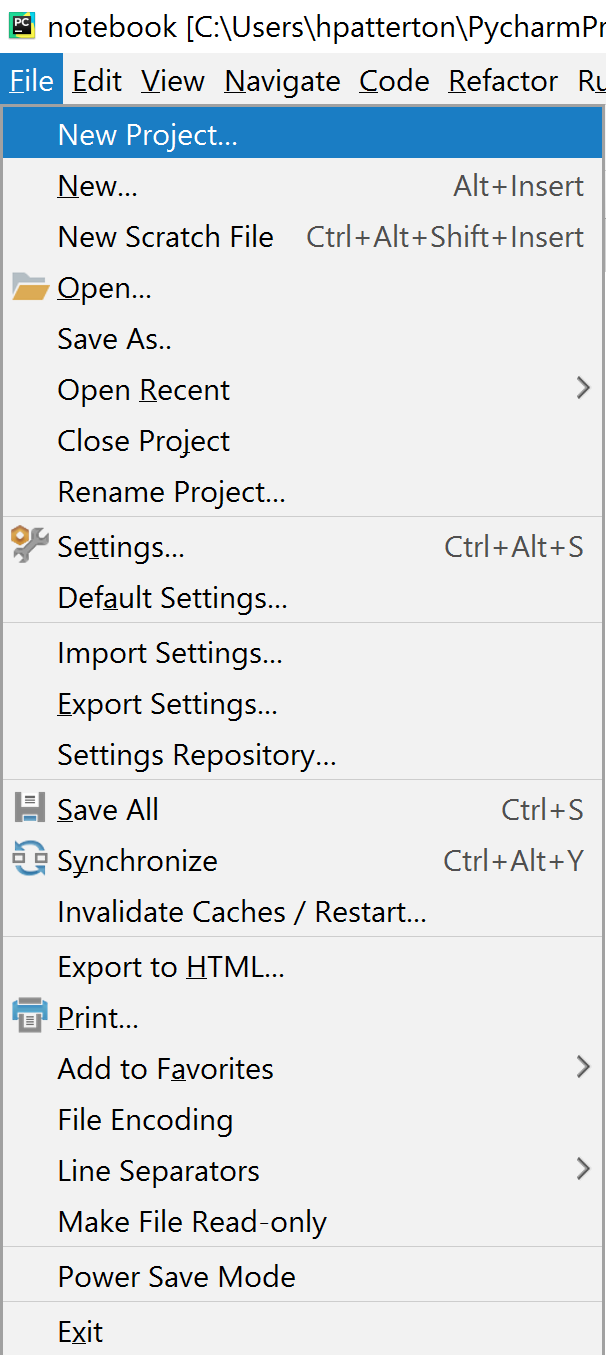
PyCharm is already installed on the Narga lab computers.

## Install PyCharm

You can download the “Community Edition” (free) of PyCharm from [www.jetbrains.com/pycharm.](http://www.jetbrains.com/pycharm) There are versions for Windows, macOS and Linux. Full installation instructions are given on the [Jetbrains website](https://www.jetbrains.com/help/pycharm/meet-pycharm.html).

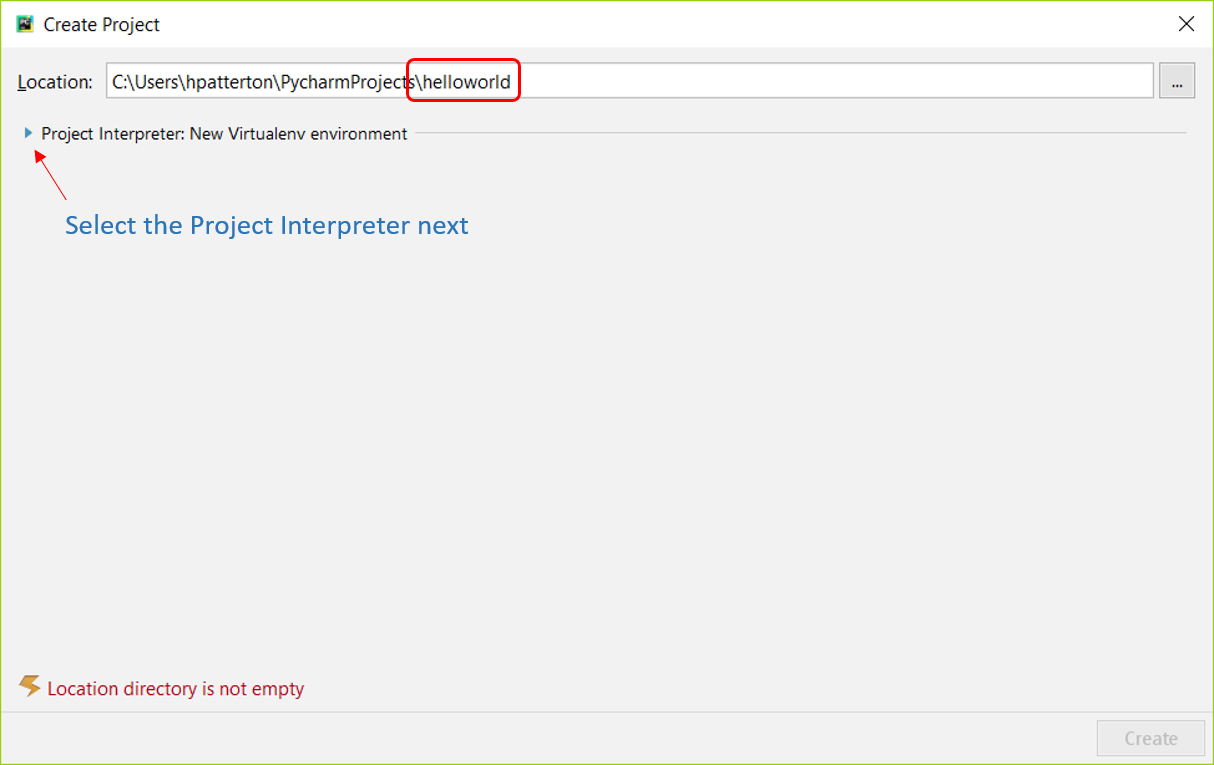
### Launch PyCharm

Find the PyCharm icon on the start menu, and launch the program.



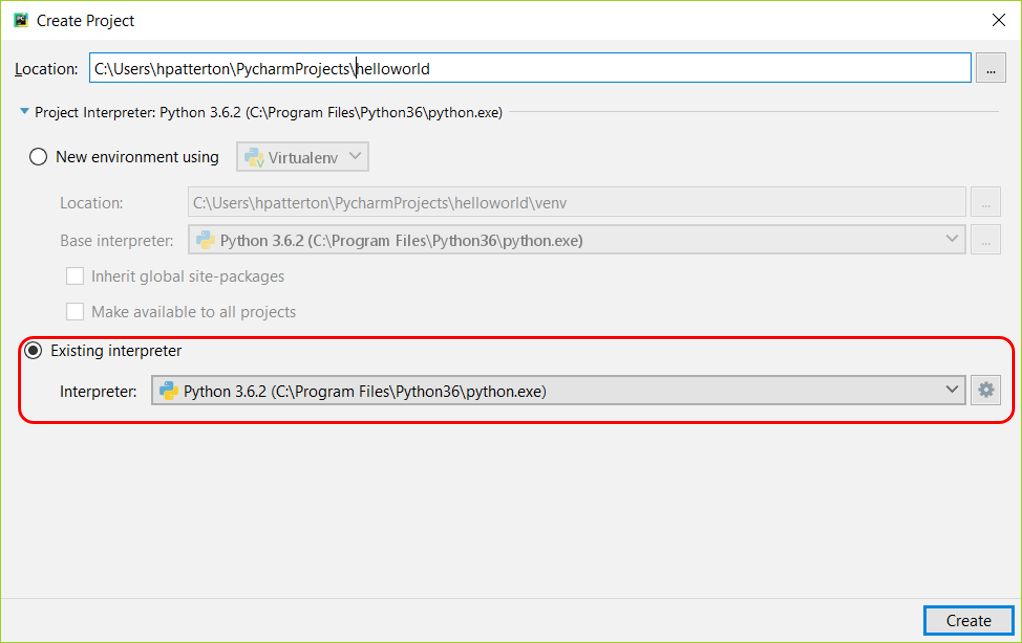
***Fig. 4.*** *Making a new project in PyCharm.*

Select **File | New Project** from the menu. In the resulting window that open, enter a project name (“helloworld” in this example) to the end of the path. Next, select the “Project Interpreter” item in the same dialog box.



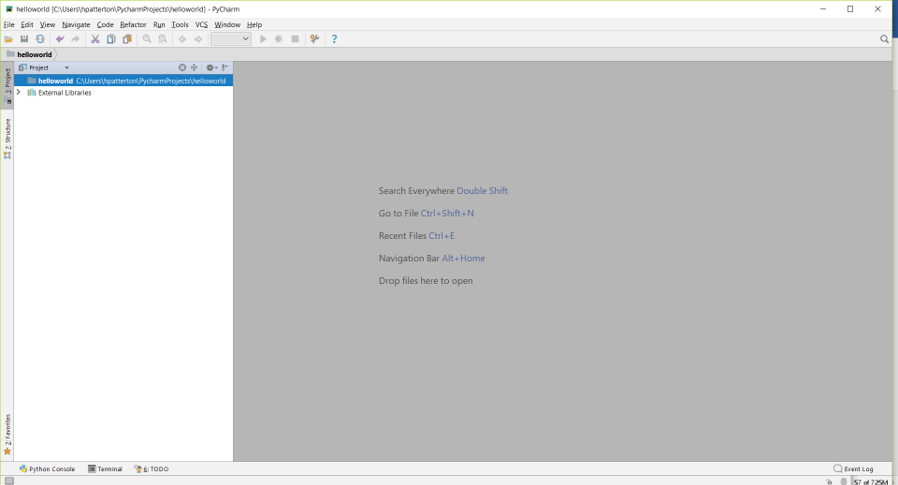
***Fig. 5.*** *Naming the new project.*

In the resulting dialog box, select “Existing Interpreter” and the most recent version of Python 3.x available on your computer in the drop-down list box.



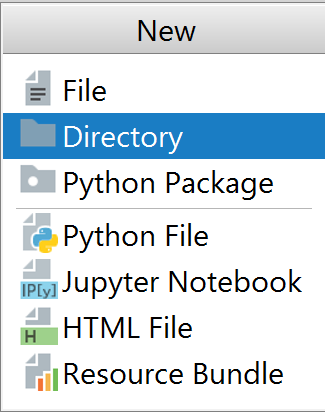
***Fig. 6.*** *Selecting the interpreter.*

The PyCharm program should now look like this. The left panel is the Project panel, and will contain a list of projects that you are working on. Currently it should have a single entry *helloworld*.



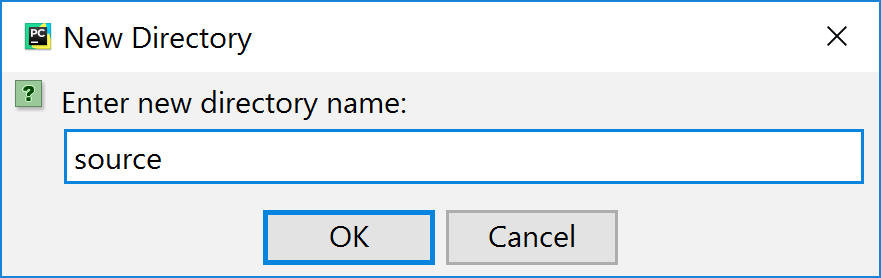
***Fig. 7.*** *The project window.*

Click on **File | New…** on the menu, and then select **Directory**.



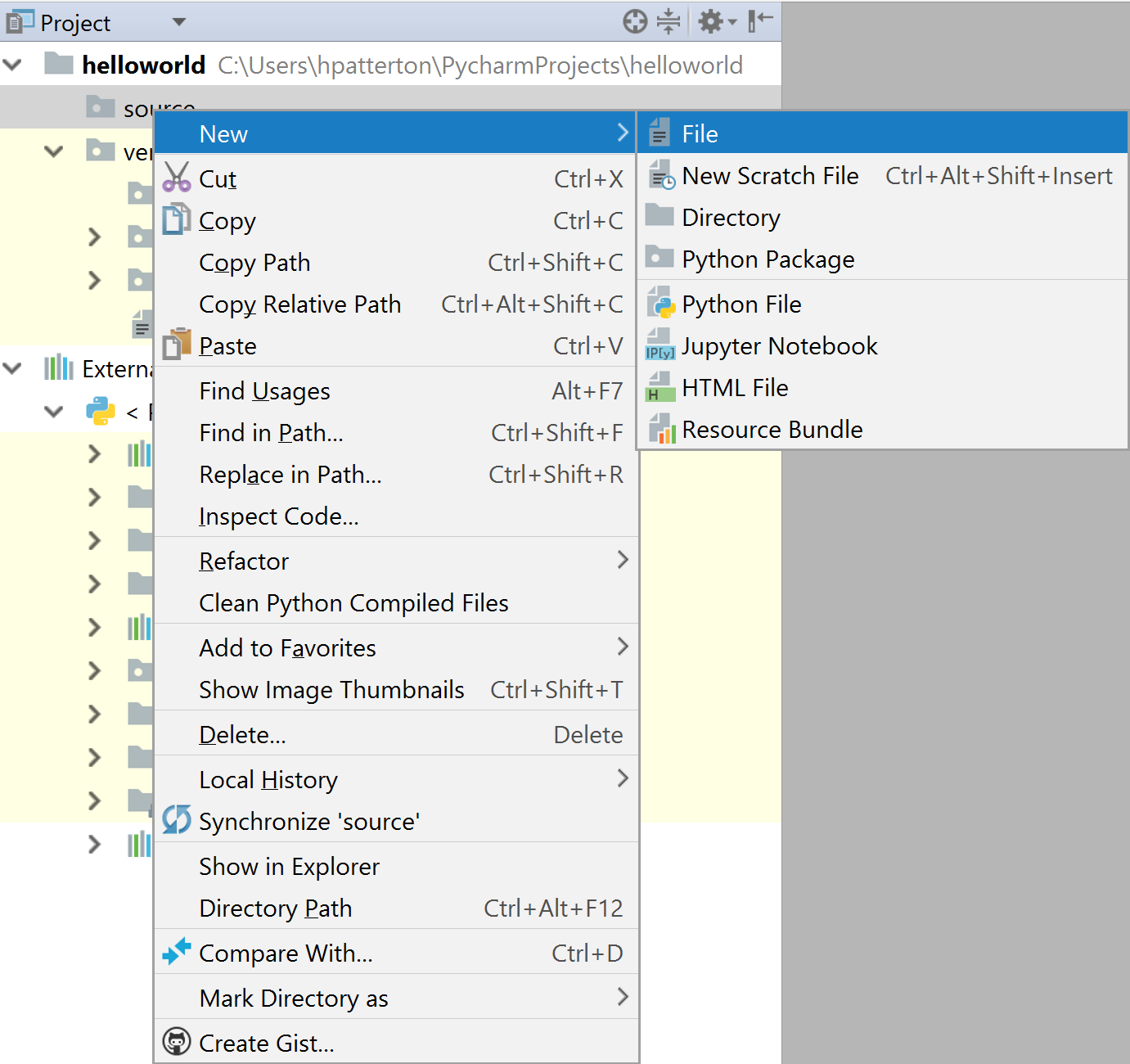
***Fig. 8.*** *Making a new directory for the project code*

Enter *source* as new directory name



***Fig. 9.*** *Naming the new directory.*

Right click on the source directory icon that appeared in the Projects panel, and select **New** and **File** from the pop-up menu (you can also select **New** and **Python File**).



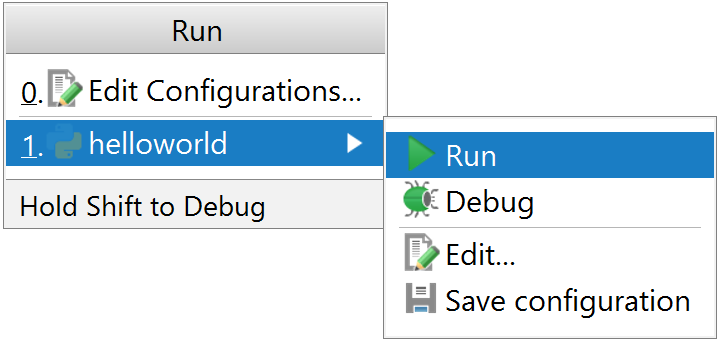
***Fig. 10.*** *Making a new code file.*

Call the new Python file *helloworld.py* in the resulting dialog window. It is good practice to call the main code file of your program the same name as your program. Python files that contain text code should always be named with the *.py* extension. If you select *Python File* from the dialog box, the *.py* extension is automatically added. The new *helloworld.py* file will immediately open in the editor panel of PyCharm.

In the *helloworld.py* editor window, type the text:

print("Hello world!")

Select Run and Run from the menu, and Run and Run from the resulting dialog windows



***Fig. 11.*** *Run* helloworld.py*.*

A new window will appear at the bottom of the PyCharm editor, the **terminal window**, and the following text should be displayed in this window:

C:\Users\hpatterton\PycharmProjects\helloworld\venv\Scripts\python.exe C:/Users/hpatterton/PycharmProjects/helloworld/source/helloworld.py

Hello world!

Process finished with exit code 0

Note that the first line in the output is the path to the Python interpreter program, followed by the path to the *helloworld.py* file, which was passed to the interpreter. The Python interpreter stepped though the text in *helloworld.py*, line by line, interpreting and executing each line until it runs out of lines. In the case of *helloworld.py* this did not take long, since there was only one line. The program produces the output “Hello world!”. The interpreter then reports that it has finished with exit code 0. This means that there were no errors.

The command *print()* is an example of an **built-in function**. There are many commands that are available in Python to perform important, general tasks.

You can save your file *helloworld.py*, and close the Project if you wish. You can open the project again to continue to work on it immediately, or the next time that you launch PyCharm.

## Comments in code

The Python interpreter ignores all text following the “#” character up to the next newline character. Thus you can add notes or comments to yourself in your code by starting the comment with a “#” character. For example:

# The section of code below reads the data from file

or

print(“Hello world!”) # This line prints a message for the user

Unlike the “/\*” and “\*/” pair in C++, Python does not have a **multi-line** comment mark. Thus you have to start each line of a comment block with a “#” character, if you want to add multiple consecutive lines of comment. Note that in PyCharm you can select a block of code, and then insert a “#” at the start of each selected line from the menu **Code > Comment with Line Comment.** If you select one or more lines starting with a “#”, you can remove (toggle) the “#” character (uncomment the code) by again selecting **Code > Comment with Line Comment** from the menu.

**Use comment lines** in your code. It makes reading and **understanding the code** much easier, especially if another programmer has to add to code that you wrote.

## Variables

A variable is the name for **an object** that can contain a **value** or information such as **text**. In other languages such as C++ that require strict typing, you must **declare** a variable before you can assign a value to it. The compiler thus knows what types of values may be assigned to a variable. In Python, the type of a variable is determined by the interpreter from the value that is **assigned** to a variable. Thus, in Python, there is no need to declare a variable; you **simply assign a value to it**. For instance, if you want to assign the value 3 to the variable a, simply write:

a=3

Make a new PyCharm project called *variables*, add a file called *variables.py*, and enter the following lines of code into the editor window:

a=3  
print(a)

Select **Run** and **Run** from the menu, and then **variables** and **Run** from the pop-up menu. The output is produced in the terminal window (for brevity we ignore the path to the interpreter and the code file, as well as the exit code):

3

In the remainder of this manual we will indicate the output of a print() statement in the terminal window as a comment on **the same line of code** to improve readability, as in:

print(a) # 3

In this case, we have set a to an **integer** (whole number). We can also set a to a **float** (a number with a fraction):

a=2.5  
print(a) # 2.5

Variable names need not be single letters. It can be alphanumeric (letters or numbers) and upper or lower case. The only exception is that variable names **cannot start with a number.**

abc123 = 5 # this is OK

1a = 3 # this is not allowed

It is good coding practice to use variable names that **describe the data that they contain**, for example

number\_of\_sequences = 200 # underscore helps to increase readability

numberofsequences = 200 # difficult to read

NumberOfSequences = 200 # difficult to read

remaining\_iterations = 150

Descriptive variable names make it easier for other coders to understand and work with your code, and it also makes it easier for you to return to your code a year after having written it, and to continue to work on it.

Variables can be used in numeric operations.

a = 2.5  
b = 3  
c = a + b  
print(c) # 5.5

A variable is equal to the last value assigned to it:

a = 2  
a = 5  
print(a) # 5

## Operators

The following operators are available in Python

***Table 3. Python 3 operators***

|  |  |  |
| --- | --- | --- |
| **Operator** | **Use** | **Example1** |
| + | Adds values on either side of the operator | a + b = 7 |
| - | Subtract right-hand operand from left-hand operand | a - b = 3 |
| \* | Multiplies values on either side of the operator | a \* b = 10 |
| / | Divides left-hand operand by right-hand operand | a / b = 2.5 |
| % | Modulo. Returns remainder after division of left-hand operand by right-hand operand. | a % b = 1 |
| \*\* | Exponent | a \*\* b = 25 |
| // | Integer division/Floor division. The fraction is discarded after division. If either operand is negative, the answer is rounded away from zero (floored). | a // b = 2  -a // b = -3 |
| 1a = 5 and b = 2 | | |

The operator precedence is **as generally used in mathematics**:

***Table 4. Operator precedence.***

|  |  |
| --- | --- |
| **Precedence** | **Operator** |
| 1 | \*\* |
| 2 | \*, /, %, // |
| 3 | +, - |
| 4 | <=, <, >, >= (comparison) |
| 5 | ==, != (equality) |
| 6 | =, %=, /=, //=, -=, +=, \*=, \*\*= (assignment) |

## Equality and assignment operators

The <=, <, > and >= operator is used to compare the magnitudes of the left-hand and right-hand operands, returning *True* if the magnitudes matches the relationship tested by the operator. The equality operators == and != (not equal) functions in the same way. The value *True* or *False* can be assigned to a variable known as a Boolean type variable (bool), named after the English mathematician George Boole, who contributed important work to algebraic logic.

**Table 5. Equality operators**

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| < | Less than |
| <= | Less than or equal |
| > | Greater than |
| >= | Greater than or equal |
| == | Equal |
| != | Not equal |

The assignment operators, in addition to =, are shorthand operators for **assignments**.

***Note***

= is used to assign a value to a variable as in a = 1. The **equality operator** == is used to **test value equality**, such as in the statement a == 1. If the value of a is 1, the statement a == 1 returns *True*, otherwise it returns *False*. **Try not to confuse assignment = with equality ==. It is the biggest source of bugs in noob code.**

**Table 6. Assignment operators**

|  |  |  |
| --- | --- | --- |
| **Assignment operator** | **Shorthand use** | **Meaning** |
| %= | a%=b | a = a%b |
| /= | a/=b | a = a/b |
| //= | a//=b | a = a//b |
| -= | a-= b | a = a - b |
| += | a+=b | a = a + b |
| \*= | a\*=b | a = a\*b |
| \*\*= | a\*\*=b | a = a\*\*b |

The += operator is very useful if you want to increment a variable by one. If you write a loop to repeat a set of instructions (see later), and need to increment the variable a by one every time you complete the loop, you can simply write

a += 1

This is shorthand for a = a + 1

This may look strange. **It is not an algebraic expression**. It is Python code. What the interpreter does is take the current value of a in memory, add 1 to that value, and then write the incremented value of a to the same memory location. The incrementing of variables is **extensively used** in code.

Note that **spaces** surrounding operators are **irrelevant.**

a + b is the same as a + b is the same as a+b.

**Spaces** are, however, **critical** to indicate the **start and end positions** of **loops** or functions (more about that later).

You can explicitly cast a variable to a desired type by making use of *float()* or *int()* to ensure you are working with what you want to be working with:

a = 2  
b = float(a)  
c = int(b)  
print(a) # 2  
print(b) # 2.0  
print(c) # 2

## Bitwise and logical operators

There are several bitwise and logical operators that are very handy in programming. A bitwise operator is an operator that is applied to bits.

***Table 7. Bitwise operators***

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | **and** | **or** | | **exclusive or** | **not1** | |
| **Symbols in Python** | | | | | |
| **Bit 1** | **Bit 2** | **&** | | **|** | **^** | | **~** |
| 0 | 0 | 0 | | 0 | 0 | | 1 |
| 1 | 0 | 0 | | 1 | 1 | | 0 |
| 0 | 1 | 0 | | 1 | 1 | | 1 |
| 1 | 1 | 1 | | 1 | 0 | | 0 |
| 1*not* is a unitary operator. It works on a single operand (here bit 1) | | | | | | | |

These bitwise operators can obviously also be applied to series of bits found in bytes and in wider binary numbers:

11001101 (=205)

**OR** 11000011 (=195)

\_\_\_\_\_\_\_\_

11001111 (=207)

In Python logical operators work like bitwise operators, except that it is applied to expressions that return True or False values. Note that in Python, unlike C++, the logical operators are not abbreviated as &&, || etc., but as the words **and**, **or** etc.

***Table 8. Logical operators.***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Condition 1** | **Condition 2** | **and** | **or** | **not1** |
| False | False | False | False | True |
| True | False | False | True | False |
| False | True | False | True | True |
| True | True | True | True | False |
| 1not is a unitary operator, applied to a single condition (condition 1 here) | | | | |

## Data Types

### Strings

A string is a collection of one or more characters, and is enclosed in single or double quotes when assigned. It does not matter whether you use single or double quotes – choose a convention and stick to it.

a = "This is a string"  
print(a) # This is a string

You can “add” (concatenate) strings:

a = **"**This is a string**"**b = " and another string"c = a + b  
print(c) *# This is a string and another string*

You can also ‘repeat’ a string:

my\_string=**'bla'**\*5  
print(my\_string) *# blablablablabla*

A string is **immutable**. That means that once a string literal (characters enclosed in quotation marks) has been assigned to a variable, **the string cannot be modified**. However, the variable can be **reassigned**. The following is perfectly legal:

a = "If she weighs the same as a duck..."  
a = a + " she's made of wood."print(a) *# If she weighs the same as a duck... she's made of wood.*

There are many functions that can be applied to strings.

For instance to convert all the letters in a string to uppercase, use the name\_of\_string.upper() function. The way is which this is written may appear odd at first. The reason why the function is written after the variable name using a point, followed by the function name, is because the function is a member function or method of the string class, and our specific string is an instantiation of that string class. All member functions are called with a point notation relative to the instantiated class. If this makes little sense now, do not worry, we will delve into classes and objects later.

a = **"This is a string"**a = a.upper()  
print(a) # THIS IS A STRING

We can also find the number of letters in a string (or members in an iterable object – see later):

a = "So why do witches burn?"  
print(len(a)) *# 23*

***Table 9. String functions that are frequently used. See the Python documentation at*** [***https://docs.python.org/3/***](https://docs.python.org/3/) ***for a full list of functions.***

|  |  |
| --- | --- |
| **Methods with Description** | **Methods with Description** |
| capitalize() | Capitalizes first letter of string |
| count(str, beg= 0,end=len(string)) | Counts how many times str occurs in string or in a sub-string of string if starting index beg and ending index end are given. |
| endswith(suffix, beg=0, end=len(string)) | Determines if string or a sub-string of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise. |
| expandtabs(tabsize=8) | Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided. |
| find(str, beg=0 end=len(string)) | Determine if str occurs in string or in a sub-string of string if starting index beg and ending index end are given returns index if found and -1 otherwise. |
| isalnum() | Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise. |
| isalpha() | Returns true if string has at least 1 character and all characters are alphabetic and false otherwise. |
| isdigit() | Returns true if string contains only digits and false otherwise. |
| islower() | Returns true if string has at least 1 cased character and all cased characters are in lower-case and false otherwise. |
| isnumeric() | Returns true if a unicode string contains only numeric characters and false otherwise. |
| isspace() | Returns true if string contains only white-space characters and false otherwise. |
| isupper() | Returns true if string has at least one cased character and all cased characters are in upper-case and false otherwise. |
| len(string) | Returns the length of the string |
| ljust(width[, fillchar]) | Returns a space-padded string with the original string left-justified to a total of width columns. |
| lower() | Converts all upper-case letters in string to lower-case. |
| lstrip() | Removes all leading white-space in string. |
| maketrans() | Returns a translation table to be used in translate function. |
| max(str) | Returns the max alphabetical character from the string str. |
| min(str) | Returns the min alphabetical character from the string str. |
| replace(old, new [, max]) | Replaces all occurrences of old in string with new or at most max occurrences if max given. |
| rfind(str, beg=0,end=len(string)) | Same as find(), but search backwards in string. |
| rindex( str, beg=0, end=len(string)) | Same as index(), but search backwards in string. |
| rjust(width,[, fillchar]) | Returns a space-padded string with the original string right-justified to a total of width columns. |
| rstrip() | Removes all trailing white-space of string. |
| split(str="", num=string.count(str)) | Splits string according to delimiter str (space if not provided) and returns list of sub-strings; split into at most num sub-strings if given. |
| splitlines( num=string.count('\n')) | Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed. |
| startswith(str, beg=0,end=len(string)) | Determines if string or a sub-string of string (if starting index beg and ending index end are given) starts with sub-string str; returns true if so and false otherwise. |
| strip([chars]) | Performs both lstrip() and rstrip() on string. |
| swapcase() | Inverts case for all letters in string. |
| title() | Returns "titlecased" version of string, that is, all words begin with upper-case and the rest are lower-case. |
| translate(table, deletechars="") | Translates string according to translation table str(256 chars), removing those in the del string. |
| upper() | Converts lower-case letters in string to upper-case. |
| zfill (width) | Returns original string left-padded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| isdecimal() | Returns true if a unicode string contains only decimal characters and false otherwise. |

For example, to capitalize the first character of each word in the string “The quick brown fox jumps over the lazy dog”:

a = **"**The quick brown fox jumps over the lazy dog**"**a = a.title()  
print(a) *# The Quick Brown Fox Jumps Over The Lazy Dog*

To find the number of times that the character “o” occurs in a string:

a = "The quick brown fox jumps over the lazy dog"  
a = a.count("o")  
print(a) *# 4*

Another example: to get the complement of a DNA sequence, first generate a translation table. The translation table is simply a table that gives the ASCII code of a character (see later and Table 13), and that of the character to which it is mapped.

a = "GATC" # 71=G, 65=A, 84=T and 67=C  
b = "CTAG" # complementary to string a sequence  
trans\_table = str.maketrans(a,b) # {71: 67, 65: 84, 84: 65, 67: 71}

Now use the generated translation table to convert a DNA sequence to its complement:

c = **'GGGATATCC'**print(c.translate(trans\_table)) *# CCCTATAGG*

#### Slicing

It is often useful to select part of a string for further manipulation, or to search for the occurrence of a shorter, sub-string. Python provides this method, which is called **slicing**. Slicing is indicated by square brackets, and a start and end position. The **string index starts at 0**. In the slicing range, the **start position** is **included**, but the **end position** is **excluded**.

a = **"**She turned me into a newt!"b = a[4:8]  
print(b) *# turn*

It is also possible to indicate the **number of steps** that must be taken when slicing a string:

string[start:end:step]

For example

b = a[4:9:2] # tre

When you omit the start value in a slice, it defaults to 0, and if you omit the end value, it defaults to the length of the string.

a[:] # The quick brown fox jumps over the lazy dog

It is also possible to step in reverse. For instance, to copy a[4:9] in reverse:

a[8:3:-1] # kciuq

Can you see why the start and end values have changed? ***Tip:*** remember what is included and what is excluded.

To reverse an entire string:

a[::-1] # god yzal eht revo spmuj xof nworb kciuq ehT

## Exercises Day 1

1. The following is an example of a compiled language
   1. Python
   2. Basic
   3. C++
   4. Perl
   5. Assembler
2. The bit pattern 1000 is equal to
   1. One more than 22+21+20
   2. One more than 999
   3. 22+21+20
   4. 23+22+21+20
   5. One more than 22×21×20
3. The decimal number 45 is the following in hexadecimal notation
   1. 2D
   2. 4D
   3. DE
   4. D2
   5. C8
4. The decimal number 2303 is the following hexadecimal number, using the little-endian system:
   1. FF08
   2. 08FF
   3. 0F08
   4. 80FF
   5. FF80
5. If each memory block is composed of two bytes, and you could represent memory addressed only as two bytes, what is the largest amount of memory in kilobytes that you could address:
   1. 127KB
   2. 16KB
   3. 2KB
   4. 4KB
   5. 1270KB
6. The Python language is
   1. Interpreted
   2. Compiled
   3. Assembled
   4. Disassembled
   5. Machine code
7. The following is a legitimate comment line in Python
   1. #-=-=-=-=-=-=-=-=-=-=-=-=-=-
   2. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
   3. -#-#-#-#-#-#-#-#-#-#-#-#-#-#
   4. /////////////////////////////
   5. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
8. The following in NOT a legitimate variable name in Python
   1. 1\_myvalue
   2. my\_value\_1
   3. myvalue1
   4. M
   5. \_myvalue
9. To assign a float to a variable in Python, you must
   1. Assign a float to the variable
   2. Declare the variable as a float before assigning to it
   3. Use a special “float name” for your variable
   4. Indicate to Python that the variable is a float, by preceding it with f\_
   5. Use the variable in round brackets
10. Given a=15 and b=4, the expression 15%4 evaluates to
    1. 3
    2. 0.06
    3. 3.75
    4. 15,4
    5. 0.1500
11. The expression 5//-2 evaluates to
    1. -3
    2. 1
    3. -1
    4. 2
    5. -2
12. The +,\*\* and \* operators have the following descending order of precedence
    1. \*\*,\*,+
    2. \*,\*\*,+
    3. \*,+,\*\*
    4. \*\*,+,\*
    5. +,\*\*,\*
13. Given a=5 and b=2, the expression (a!=b) evaluates to
    1. True
    2. False
    3. 2.5
    4. -3
    5. 2
14. The expression (11001100^11110000) evaluates to
    1. 00111100
    2. 11000011
    3. 10101010
    4. 11111100
    5. 00000011
15. The expression (11001100|11110000) evaluates to
    1. 11111100
    2. 11001100
    3. 00110011
    4. 10101010
    5. 01010101
16. Given a=’ABCDDEEFF’ and b='E', the expression print(a.count(b)) evaluates to
    1. 2
    2. 9
    3. 1
    4. 4
    5. 4.5
17. Given a=’ SOUTHSEA’ the expression a.lstrip() evaluates to
    1. ‘SOUTHSEA’
    2. ‘southsea’
    3. ‘sea’
    4. ‘OUTHSEA’
    5. ‘OUTHEA’
18. Given a=’SAUSSAGE’ the expression a.split('A')) evaluates to
    1. ‘S’,’USS’,’GE’
    2. ‘SA’,’SSA’,’GE’
    3. ‘S’,’AUSS’,’AGE’
    4. ‘SAUS’,’SAGE’
    5. ’S’,’S’,’S’
19. The index of ‘R’ in ‘RABBIT’ is
    1. 0
    2. 1
    3. -1
    4. R
    5. R
20. Given a ='rabbit of caerbannog'’, a[5:11] is
    1. ‘t of c’
    2. ‘ of c’
    3. ‘t of ca’
    4. ‘c of t’
    5. ‘tofc’
21. Given a='rabbit of caerbannog', a[20:9:-1]) evaluates to
    1. ‘gonnabreac’
    2. ‘caerbannog’’
    3. ‘aerbannog’
    4. ‘caerbanno’
    5. ‘gonnabrea’
22. Given a='able was i ere i saw elba', a[::-1] evaluates to
    1. 'able was i ere i saw elba'
    2. 'ablewasiereisawelba'
    3. 'able was i erble was i'
    4. ‘’
    5. ‘elba’
23. Given a = ‘ABCDEFGHI’, a[6::-1] evaluates to
    1. GFEDCBA
    2. GHI
    3. IHG
    4. IHGFED
    5. IHGFE
24. Given a = "And finally, monsieur, a wafer-thin mint", a.rindex('m') evaluates to
    1. 36
    2. 13
    3. 3
    4. 39
    5. 4
25. Given a = "Iwanthimfightingwabidwildaninmalswithinaweek", a.isalpha() evaluates to
    1. True
    2. False
    3. Generates an error
    4. 1
    5. 44
26. Given A='It\'s not easy to pad these python files out to 150 lines, you know', the statement B=A.replace('python files', 'norwegian blues', 0) will evaluate to:
    1. ‘It's not easy to pad these python files out to 150 lines, you know'
    2. ‘It's not easy to pad these norwegian blues out to 150 lines, you know'
    3. ‘python files’
    4. ‘norwegian blues’
    5. ‘It's not easy to pad these python files out to 150 lines'

### Lists, tuples, sets and dictionaries

#### Lists

Since strings are immutable, you cannot use a string if you want to change some of its letters:

a = “This isa string”  
a[13] = “o” *# this is not allowed*

You can **re-assign** a string variable

a=a[10:16] *# a = “string”*

However, it is often useful in bioinformatics to have a long list of values where you can randomly select one or more and change their values, i.e., a **mutable** data type. This is where the compound data type *list* comes in. A list is a collection of items (numbers, characters, strings, etc.) and is assigned in square brackets and separated by commas.

my\_list = [1,"b",3,"zzz",5]

Although a list can contain a mixture of data types as shown above, it is more usual to use a single data type

my\_list = ["A","B","C","D","E"]

***Note***

Python does not have the data type *character* (char) that is found in C++. Single letter strings are used to represent characters.

Like strings, lists can be added and repeated:

my\_list1 = [1,2,3]  
my\_list2 = [4,5,6]  
my\_list3 = [0]\*3 *# [0, 0, 0]*  
print(my\_list1+my\_list2+my\_list3) *# [1, 2, 3, 4, 5, 6, 0, 0, 0]*

It is possible to substitute entries in lists using slice notation

my\_list = ["A","B","C","D","E"]

my\_list[2:] = ["X","Y","Z"] *# ['A', 'B', 'X', 'Y', 'Z']*

The list produced is simply expanded if the inserted list exceeds the space specified in the splice:

my\_list[2:3] = ["X","Y","Z"] *# ['A', 'B', 'X', 'Y', 'Z', 'D', 'E']*

Assigning an empty list to a slice region of a list, overwrites the elements:

my\_list[2:4] = [] *# ['A', 'B', 'E']*

A string can be converted to a list with the built-in function *list()*:

my\_string = "This is a string"

my\_list = list(my\_string)

*# ['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g']*

There are several other functions that can also be applied to lists. The more frequently used ones are listed in Table 10.

***Table 10. Functions applicable to lists.***

|  |  |
| --- | --- |
| **Method** | **Description** |
| list() | creates a list |
| len() | returns length (number of entries) of the list |
| max() | returns largest element |
| min() | returns smallest element |
| slice() | creates a slice specified by range() |
| sum() | sum of list (if integers or floats) |
| append() | add single element to the list |
| extend() | add elements of a list to another list |
| insert() | inserts element(s) |
| remove() | removes element from the list |
| index() | returns smallest index of element in list |
| count() | returns occurrences of element in a list |
| pop() | removes element at given index |
| reverse() | reverses a list |
| sort() | sorts elements of a list |
| copy() | returns shallow copy of a list |
| clear() | removes all Items from the list |

#### List comprehension

We have seen how to assign a sequence of integers or strings or another variable type to a list. There is a very elegant way, called **list comprehension**, to assign values to a list, but we need to get a little bit ahead of ourselves to see how this is done. In mathematics one can define the members of a set as {n**N**:n<100}, meaning *n* is a member of the group of natural numbers, smaller than 100, i.e., 1-99. This notation is known as a **set comprehension**: a succinct mathematical description of the members of the set. This same term is used to define the members of a list in Python:

my\_list = [x **for** x **in** range(1,10)] *# list comprehension*  
print(my\_list) *# [1, 2, 3, 4, 5, 6, 7, 8, 9]*

We have not yet discussed *for-loops*, so the code above may appear obscure. What it essentially does is equate the variable x to the value of x generated in the *for-loop*: for x in range(1,10). Range(1,10) returns a sequence of integers 1 to 9 assigned to x in the *for-loop*, and the x (or x in an expression) preceding the *for-loop* is assigned this number. A list comprehension is basically an expression followed by a *for-loop*, followed by zero or more additional *for-loops* or *if* statements.

my\_list = [x-2 **for** x **in** range(1,10) **if** x < 5]  
print(my\_list) *# [-1, 0, 1, 2]*

The variable x is set to the sequence 1 to 9 only if the value is less than 5, and 2 is subtracted from this value before it is assigned as a member of the list my\_list. List comprehension is a very solid method to define lists, so if the code does not quite make sense now, bookmark it, and return to this section after we have discussed *for-loops* and *if* statements.

#### Tuples

A tuple is a data type assigned to a variable, where the data types can be mixed. This is like a list, but, unlike lists, tuples are **immutable**.

my\_tuple = 12345,67890,**"a string"**print(my\_tuple) *# (12345, 67890, 'a string')*

My\_tuple[1] = 10 *# not allowed*

Note that tuples are enclosed in **round brackets**. Lists are enclosed in **square brackets**. Tuples can also be **unpacked** into individual variables:

my\_tuple = 12345,67890,**"a string"** *# tuple packing*x,y,z = my\_tuple *# tuple unpacking*print(x,y,z) *# 12345 67890 a string*

Note that variable z is a string

print(type(z)) *# <class 'str'>*

A tuple is especially handy if you have functions that return more than one value. More about functions later.

If you are interested in only the first couple of values in a tuple, it is possible to unpack only the ones you are interested in and to discard the rest with the **\* notation**.

my\_tuple = (1,2,3,4,5)  
a,b,\*not\_wanted = my\_tuple  
print(a,b) *# 1 2*print(type(not\_wanted),\*not\_wanted) *# <class 'list'> 3 4 5*

The first two values in my\_tuple is assigned to a and b, and the rest to a list, not\_wanted. Can you use the \* notation to unpack the first two and last two values in a multi member tuple?

#### Generator expressions

Tuples have a quick method to generate a sequence of tuples, called a **generator expression**, **like list comprehension**. The following code generates a tuple of tuples.

my\_tuple = tuple((x,y) **for** x **in** range(1,10) **for** y **in** range(1,10) **if** x < 3 **if** y < 4)  
print(my\_tuple) *# ((1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3))*

You can also use generator expressions to create a list of tuples.

my\_list = [(x,y) **for** x **in** range(1,10) **for** y **in** range(1,10) **if** x < 3 **if** y < 4]  
print(my\_list) *# [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)]*

#### Sets

A set is related to a list, but **all elements are** **unique**, and the members are **strings**. A set is generated with the set() function, or assigned with curly brackets {}

a = set(**"**abcde**"**) *# {'e', 'b', 'd', 'c', 'a'}*b = {**"**fghij**"**} *# {'fghij'}*

Note that the *set()* function takes each character in the supplied string as an element, whereas the {} assignment assigns the whole string as one element.

One can perform a number of operation on the sets to test differences. (Refer back to **Table 7** for bitwise operators).

a = set(**"wellheisprobablypiningforthefjords"**)  
b = set(**"lookwhydidhefallflatonhisbackthemomentigothimhome"**)  
print(sorted(a)) # ['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'l', 'n', 'o', 'p', 'r', 's', 't', 'w', 'y']  
print(sorted(b)) # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 's', 't', 'w', 'y']  
print(sorted(a-b)) # ['j', 'p', 'r']  
print(sorted(a|b)) # OR ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'w', 'y']  
print(sorted(a&b)) # AND ['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i', 'l', 'n', 'o', 's', 't', 'w', 'y']  
print(sorted(a^b)) # XOR ['c', 'j', 'k', 'm', 'p', 'r']

#### Dictionaries

A dictionary is a compound data type where each member is composed of two items: a **key** and a **value**. The key must be **immutable**, such as a string, a number or a tuple. A dictionary is defined by specifying member pairs in curly brackets.

my\_dictionary = {‘entry 1’:1, ‘entry 2’:2}

print(my\_dictionary) *# {'entry 1': 1, 'entry 2': 2}*

Some other examples:

my\_dictionary = {1:1, 2:2} # {1: 1, 2: 2}

my\_dictionary = {1:'value 1', 2:'value 2'} *#{1: 'value 1', 2: 'value 2'}*

You can also use a list of strings, since each string is immutable:

my\_list = ["name1","name2"]

my\_dictionary = {my\_list[0]:"1",my\_list[1]:2} *#{'name1': '1', 'name2': 2}*

The usefulness of a dictionary is that you can easily obtain the value associated with the key:

my\_dictionary = {'key 1':"100",'key 2':"200"}

print("value =",my\_dictionary['key 1']) *#value = 100*

or, slightly more complex, using a list of keys:

my\_list = ["name1","name2"]

my\_dictionary = {my\_list[0]:"1",my\_list[1]:2}

print("Key =",my\_list[0],"Value =",my\_dictionary[my\_list[0]])

*# Key = name1 Value = 1*

Each key is associated with a single value, and keys are not duplicated. If you try to add two keys with the same name but associated with different values, **the last value is used** for the single, unique key.

You can add entries to a dictionary by simply assigning a value to a new key:

my\_dictionary['name3']=300

print(my\_dictionary) *# {'name1': '1', 'name2': 2, 'name3': 300}*

Single entries or the entire content of dictionaries can be cleared, or an entire dictionary can be deleted:

del my\_dictionary['name1'] # clear entry ‘name1’

my\_dictionary.clear() # clear all entries

del my\_dictionary # delete my\_dictionary

There are several useful functions that can be applied to dictionaries:

**Table 11. Dictionary functions.**

|  |  |
| --- | --- |
| **Function** | **Description** |
| clear() | Removes all elements of the dictionary |
| copy() | Returns a shallow copy of dictionary |
| fromkeys() | Create a new dictionary with keys from seq and values *set* to *value*. |
| get(key, default=None) | For *key* key, returns value or default if key not in dictionary |
| items() | Returns a list of (key, value) tuple pairs |
| keys() | Returns list of dictionary’s keys |
| setdefault(key, default = None) | Similar to get(), but will set dict[key] = default if *key* is not already in dict |
| update(dict2) | Adds dictionary *dict2*'s key-values pairs |
| values() | Returns list of dictionary's values |

***Note***

A dictionary **does not maintain its keys or values in a specific order**, so do not assume that when you recover a list of keys or values, that the order will be identical between two calls. You can use *OrderedDict()* or *SortedDict()* available in the modules *collections* and *sortedcontainers*, respectively, if you want a sorted dictionary.

## Data Input and Output

### Input from the keyboard

The function *input()* reads input from the standard input, which is generally the keyboard. It is typically used as follows:

a = input()

The function “waits” until the **Enter** key is pressed, and then passes whatever was typed as **a string** to the variable a. Remember to click in the terminal window of PyCharm to **ensure that the terminal window has focus** when testing *input()*. You can also add a prompting string to the function:

a = input(“Enter a number:”)

Irrespective of the prompting string, the function *input()* always returns **a string**. For instance, entering a dictionary as input returns the input type as string:

a = input("Input:")

print(a,type(a))

*# Input:{'name1':1,'name2':2}*

*# {name1':1,'name2':2} <class 'str'>*

If you want a specific data type, an integer for instance, you can set the type of the data by **casting**. For instance, you use the function *int()* to cast to an integer:

a = int(input("Enter a number:"))

print(a,type(a))

*# Enter a number:10*

*# 10 <class 'int'>*

### Reading data from a file

To read a file in Python you need to specify the full file path and the mode in which the file should be opened. This is provided by using the function *open(‘file path’, ‘mode’)*. The mode can be ‘r’ reading, ‘w’ writing, ‘a’ appending (writing data to the end of an existing file) and ‘r+’, both reading and writing. If the mode is omitted, ‘r’ is used by default. You can also specify newline=’’, ‘\n’, ‘\r’ or ‘\n\r’. The newline parameter defaults to none (‘’), meaning that a universal newline character will be used: ‘\n’, ‘\r’ and \n\r’ will all be recognized as a newline.

Calling the function returns a **file object**. The file object is used to read from the file:

my\_file = open('C:\\Users\\hpatterton\\Documents\\my\_file.txt','r')

my\_data = my\_file.read()

print(my\_data) *# I will not buy this record, it is scratched.*

my\_file.close()

A couple of points here. The file path directories are separated by **double** forward slashes ‘\\’. This is because string can contain what is known as **escape characters**. For instance, I can represent a tab character in a string as **‘\t’**, or a newline as **‘\n’** as in ‘I will not buy this record\tit is scratched\n’. This comes from the days of **teletext**, when a way had to be found to format a linear stream of transmitted text into lines and paragraphs etc. The solution was to use “escape” characters in the text. The forward slash basically tells Python to use the next character as an escape code. If you mean to use the forward slash character itself in a string, you must ‘escape’ it, by using a double slash. This is why the path string above has double forward slashes. ccccccc

So, Python opens the file ‘*my\_file.txt*’ and associates it with the file object *my\_file*. The file object (see later) has many methods that can manipulate the file object, like the string object we saw earlier. We use the object methods to read the contents of the file and to close the file. You will need to re-open a closed file (get another file object) if you want to read from it again. It is good practice to close a file immediately when you are done reading (or writing) to avoid engaging system resources unnecessarily. The function *read()* will attempt to read the entire contents of the file into memory. You must make sure that the specific computer that you use has sufficient memory to accommodate the contents of the file, and that the operation succeeded.

Apart from reading the entire contents of the file, you can also read it line by line. In text mode (the default) the *readline()* function will read a line of characters until a newline character is encountered, and then return the line of characters **with** training newline.

my\_data = my\_file.readline()

print(my\_data) # I will not buy this record, it is scratched

my\_data = my\_file.readline()

print(my\_data) # My hovercraft is full of eels

Unless you are interested in only reading the first line or the first few lines in a file, the code construct above is extremely cumbersome. Iteration to the rescue! You have probably realized by now that variables in Python are all objects, and that some objects have methods. Some objects have the property that they are *iterable* (see *Containers and Iterations* later). That means that one can request an object to provide the values from a list of values one at a time, until the list is exhausted. A file object is *iterable*. Thus, we can use a *for-loop* to obtain each line in turn:

**for** line **in** my\_file:  
 print(line)

A slightly more contorted way is using a **while** loop. ***Note:*** *getline()* returns a line with a trailing newline (“\n\r” on Windows, “\n” on Linux). If the last line in the file does not end with a newline, the last line will be returned by *getline()* without a trailing newline. If you read a next line when the file has been fully read (the file pointer points past the end of the file), *getline()* return an empty string. Thus, you can read the file line by line in a while loop until an empty string is returned:

line = my\_file.readline()

while(line !=''):

print(line)

line = my\_file.readline()

my\_file.close()

You can also use the *readlines(hint=n)* (note the plural) function to read several lines at a time.  *Readlines(hint=n)* will continue reading lines until the sum of all characters or bytes read exceed *n*, and then return the lines that was read.

***Note***

The most efficient way to read a file is in a single chunk (my\_file.read()), and then using a *for-loop* to retrieve the lines iteratively, if this is meaningful for the format, such as for an alignment file or fastQ sequence data file (see **Appendix A** for file formats). If it is a fastA file of individual chromosome sequence data, you may want to combine the data for each chromosome into a string or a list, for subsequent analyses and manipulations.

**Table 12. File functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| flush() | Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects. |
| fileno() | Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system. |
| next() | Returns the next line from the file each time it is being called. |
| read([size]) | Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). |
| readline([size]) | Reads one entire line from the file. A trailing newline character is kept in the string. |
| seek(offset[, from\_where]) | Sets the file's current position  (from where = 0 (beginning), 1 (current position), or 2(end)) |
| tell() | Returns the file's current position |
| truncate([size]) | Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size. |
| write(str) | Writes a string to the file. There is no return value. |
| writelines(sequence) | Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. |

### Manipulating bits and bytes in Python

Python provides two built-in functions to deal with binary data: *bytes()* and *bytesarray()*. The data type returned by *bytes()* is **immutable**, and that of *bytesarray()* is **mutable**.

You can pass a single value or a list of values to the *bytes()* function:

my\_list = [1,2,3,5,7,11,13,255]  
my\_bytes = bytes(my\_list)  
print(my\_bytes) # b'\x01\x02\x03\x05\x07\x0b\r\xff'  
print(my\_bytes[6]) # 13

my\_bytes[2] = 4 # Cannot do this; immutable

***Note:*** the size of each value must be 0-255.

my\_list = [1,2,3,5,7,11,13,255]  
my\_bytes = bytearray(my\_list)  
my\_bytes[7] = 17  
print(my\_bytes) # bytearray(b'\x01\x02\x03\x05\x07\x0b\r\x11')

We discuss the meaning of b'\x01\x02\x03\x05\x07\x0b\r\x11' in the next sections.

### Representation of Characters

A string is represented as a series of numbers, which is translated using a character map. The simplest mapping scheme is the **ASCII** (American Standard Code for Information Interchange)map, where characters are assigned a value between 0 and 127, which can be represented by 7 bits. The ASCII code was originally developed by Bell Laboratories in the 1950s to allow communication between telegraph machines. The upper and lower-case characters, numbers, and some punctuation marks are represented, as well as **non-printable control characters** used to format a message. Most of the non-printable codes are now obsolete. Later character maps were based on and extended the original 7 bit ASCII set. For instance the extended ASCII set doubles the size of the map by using 8 bits to represent characters, and includes non-English characters, mathematical symbols, block lines and so forth (see [en.wikipedia.org/wiki/Extended\_ASCII](https://en.wikipedia.org/wiki/Extended_ASCII)). A modern map is the UTF-8 (Unicode Transformation Format – 8-bit) map which can represent 1,112,064 characters using up to 4 bytes (see [en.wikipedia.org/wiki/UTF-8](https://en.wikipedia.org/wiki/UTF-8)). UTF-8 is currently the most widely used map. Python can translate a value to the corresponding character using a map using the internal function *chr()*. A character can be translated to its numerical value using *ord()*. Both use the UTF-8 map which is the Python default.

print(chr(99)) # c

print(ord('c')) # 99

***Table 13. The ASCII data table.***

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code** | **Meaning** | **Code** | **Meaning** | **Code** | **Meaning** | **Code** | **Meaning** |
| 0 | NULL | 32 | space | 64 | @ | 96 | ` |
| 1 | start of header | 33 | ! | 65 | A | 97 | a |
| 2 | start of text | 34 | " | 66 | B | 98 | b |
| 3 | end of text | 35 | # | 67 | C | 99 | c |
| 4 | end of transmission | 36 | $ | 68 | D | 100 | d |
| 5 | enquiry | 37 | % | 69 | E | 101 | e |
| 6 | acknowledgement | 38 | & | 70 | F | 102 | f |
| 7 | bell | 39 | ' | 71 | G | 103 | g |
| 8 | back space | 40 | ( | 72 | H | 104 | h |
| 9 | horizontal tab | 41 | ) | 73 | I | 105 | i |
| 10 | linefeed | 42 | \* | 74 | J | 106 | j |
| 11 | vertical tab | 43 | + | 75 | K | 107 | k |
| 12 | form feed | 44 | , | 76 | L | 108 | l |
| 13 | carriage return | 45 | - | 77 | M | 109 | m |
| 14 | shift out | 46 | . | 78 | N | 110 | n |
| 15 | shift in | 47 | / | 79 | O | 111 | o |
| 16 | data link escape | 48 | 0 | 80 | P | 112 | p |
| 17 | device control 1 | 49 | 1 | 81 | Q | 113 | q |
| 18 | device control 2 | 50 | 2 | 82 | R | 114 | r |
| 19 | device control 3 | 51 | 3 | 83 | S | 115 | s |
| 20 | device control 4 | 52 | 4 | 84 | T | 116 | t |
| 21 | negative acknowledgement | 53 | 5 | 85 | U | 117 | u |
| 22 | synchronous idle | 54 | 6 | 86 | V | 118 | v |
| 23 | end of transmission block | 55 | 7 | 87 | W | 119 | w |
| 24 | cancel | 56 | 8 | 88 | X | 120 | x |
| 25 | end of medium | 57 | 9 | 89 | Y | 121 | y |
| 26 | substitute | 58 | : | 90 | Z | 122 | z |
| 27 | escape | 59 | ; | 91 | [ | 123 | { |
| 28 | file separator | 60 | < | 92 | \ | 124 | | |
| 29 | group separator | 61 | = | 93 | ] | 125 | } |
| 30 | record separator | 62 | > | 94 | ^ | 126 | ~ |
| 31 | unit separator | 63 | ? | 95 | \_ | 127 | delete |

You can translate more complex binary patterns:

my\_string = "spam"

binary\_code = my\_string.encode('utf-8')

print(binary\_code) # b’spam’

What does b’spam’ mean? Python **represents binary data as strings**, preceded by the letter ‘b’. You can also assign a **binary string** to a variable:

my\_string = b"eggs"

my\_string2 = "and spam"

print(type(my\_string)) # <class 'bytes'>

print(type(my\_string2)) # <class 'str'>

***Note***

**Binary values are displayed** as the corresponding ASCII character if the character is **printable**. **Escape sequences** such as \t or \n is shown as \t or \n, and ASCII values that map to **non-printable** characters are shown as the hexadecimal representation of the values (eg. 255 as \xff)

The character represented by each of the values are correctly decoded:

my\_string = **b"eggs"**print(my\_string.decode()) # eggs

Let’s encode the binary representation with the actual values (see Table 10)

my\_string = bytes([101,103,103,115]) # pass a list of bytes to function  
print(my\_string.decode()) # eggs

We can also pass the character values as hexadecimal values in a string

my\_string = b"\x65\x67\x67\x73"

print(my\_string.decode()) # eggs

You **encode** from a **string to a byte** object, and you **decode** from a **byte object to a string**.

We can convert between hexadecimal values and a binary string with *fromhex()*, and from a binary string to hexadecimal values with *hex()*:

print(bytes.fromhex(**'fff0 f1f2'**)) *# b'\xff\xf0\xf1\xf2'*

and

a = **b'\xff\xf0\xf1\xf2'**print(a.hex()) *# fff0f1f2*

We can specify individual bits in a byte by passing the pattern of bits as a string to the internal function *int()*, specifying that it is an integer base 2.

my\_byte = int(**'11110000'**, 2) # 240

You can also convert a integer to a binary string with *bin()*:

a=bin(240)  
print(a) *# 0b11110000*

### Writing data to a file

When writing to a file, you open the file on the disk and obtain a file object:

my\_file = open(‘full file path’,’mode’) # mode is ‘w’, ‘r+’, ‘wa’, ‘wb’

By default data is written to disk as text characters (the ASCII code equivalent of the character).

file\_path = **"C:\\Users\\hpatterton\\Documents\\my\_writefile.txt"**my\_file = open(file\_path,**'w'**)  
my\_data = str([1,2,3,4,5,6,7,8,9]) # convert list of numbers to my\_file.write(my\_data); # string  
my\_file.close()

If you looked at the raw data of the file *my\_writefile.txt* on disk using a **hex editor** such as the open source [wxHexEditor](https://sourceforge.net/projects/wxhexeditor/) (a program that allows you to inspect the bytes in a file on disk, displaying the data as both bytes in hexadecimal and ASCII format), you will see the following sequence:

5B 31 2C 20 32 2C 20 33 2C 20 34 2C 20 35 2C 20 36 2C 20

37 2C 20 38 2C 20 39 2C 20 31 30 2C 20 31 31 2C 20 31 32

2C 20 31 33 2C 20 31 34 2C 20 31 35 2C 20 31 36 5D

These are the ASCII (or UTF-8) **values** of the **characters** in the list converted to **text**.

In bioinformatics, you will often want to write **binary data** to a file. This data can either be a compressed data file, like a BAM file, an image file and so on. To write **binary** data, simply add ‘b’ to the mode. When you write binary data, the *write()* function **expects** a **byte data type**, hence the conversion of the list to bytes using the function *bytes()*.

file\_path = **"C:\\Users\\hpatterton\\Documents\\my\_writefile.dat"**my\_file = open(file\_path,**'wb'**)  
my\_data = bytes([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16])  
my\_file.write(my\_data);  
my\_file.close()

If you looked at the raw data in the file *my\_writefile.dat* using a hex editor, you will see:

01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10

This is the representation of the **number**, not the **value of the characters** that is used to represent the numbers.

To read the binary data from the disk:

file\_path = **"C:\\Users\\hpatterton\\Documents\\my\_writefile.dat"**my\_file = open(file\_path,**'rb'**)  
binary\_string = my\_file.read() # data type is bytesmy\_file.close()  
print(binary\_string) # b'\x01\x02\x03\x04\x05\x06\x07\x08\t'

The read binary string can be converted to other data types, as needed:

my\_list = list(binary\_string) # [1, 2, 3, 4, 5, 6, 7, 8, 9]my\_bytes = bytes(binary\_string) # b'\x01\x02\x03\x04\x05\x06\x07\x08\t'print(my\_bytes[8]) # 9

**Table 14. Summary of functions for converting between binary types and strings**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Input** | **Function** | **Output** | **Type** |
| int | 120 | chr() | x | str |
| str | x | ord() | 120 | int |
| str | b’11110000’ | int(string,2) | 240 | int |
| int | 240 | bin() | b’11110000’ | str |
| hex | ff 0f | bytes.fromhex() | ‘\xff\x0f’ | str |
| str | b’\xff\x0f’ | hex() | fff0 | bytes |
| str | ‘eggs’ | encode() | b’eggs’ | bytes |
| bytes | b'\x65\x67\x67\x73' | decode() | eggs | str |

## Exercises Day 2

1. A list is assigned as follows:
   1. my\_list = [1,2,3,4,5]
   2. my\_list = {1,2,3,4,5}
   3. my\_list = (1,2,3,4,5)
   4. my\_list = <1,2,3,4,5>
   5. my\_list = /1,2,3,4,5\
2. Given the list a=[1,2,3,4,5], the statement a[2:4:] will return
   1. [3]
   2. [2]
   3. [2,4]
   4. [3,4]
   5. [2,3,4]
3. Given the list a=[‘a’,’b’,’c’,’d’,’e’], the statement a[::-1] will return
   1. [‘e’,’d’,’c’,’b’,’a’]
   2. [‘e’]
   3. [‘a’,’b’,’c’,’d’,’e’]
   4. [‘a’]
   5. An error
4. Given the list a=[1,2,3,4,5], the statement a[3:5]=[4,5,6] will return a as
   1. [1,2,3,4,5,6]
   2. [1,2,4,5,6]
   3. [1,2,3,4,5]
   4. [4,5,6]
   5. [1,2,3,6]
5. The function list(“small stones”) will return the following
   1. [‘s’,’m’,’a’,’l’,’l’,’ ’,’s’,’t’,’o’,’n’,’e’,’s’’]
   2. [‘small stones’]
   3. Strings ‘small’ and ‘stones’
   4. [‘small’,’stones’]
   5. ‘small stones’
6. Given a='Nobody expects the Spanish Inquisition!', the function a.count(‘i’) returns:
   1. 4
   2. 39
   3. 5
   4. 1
   5. -3
7. A tuple is a useful data type to
   1. Return multiple values from a function
   2. Pass multiple values to a function
   3. Store data pairs
   4. Use for a look-up table
   5. Store values that frequently change
8. Given the statement a=set(‘11234’), the function print(a) will return:
   1. {'4', '2', '1', '3'}
   2. {1,1,2,3,4}
   3. {‘1’,’1’,’2’,’3’,’4’}
   4. [1,1,2,3,4]
   5. [‘1’,’1’,’2’,’3’,’4’]
9. Given the statement my\_list = ["name1","name2"], the following is proper Python code:
   1. my\_dictionary = {my\_list[0]:"1",my\_list[1]:2}
   2. my\_dictionaty={my\_list[name1]:1, my\_list[name2]:2}
   3. my\_dictionaty={my\_list[“name1”]:1, my\_list[“name2”]:2}
   4. my\_dictionaty={my\_list[name1],1:my\_list[name2],2}
   5. my\_dictionary = [my\_list[0]:"1",my\_list[1]:2]
10. Given a={‘a1’:1,’a2’:2}, the statement a.values() will return
    1. dict\_values([1, 2])
    2. dict\_items([('a1', 1), ('a2', 2)])
    3. dict\_keys(['a1', 'a2'])
    4. dict([‘a1’:1,’a2’:2])
    5. ‘a1’,’a2’
11. The dictionary function keys() will always return:
    1. Return the keys in no order
    2. Sorted, in ascending order
    3. Sorted in descending order
    4. Sorted by value, ascending order
    5. Sorted by value, descending order
12. The built-in function input() will return
    1. The input as a string
    2. A type as defined during entry
    3. Letters as letters, and number as numbers
    4. Integers
    5. The return type can be defined by a parameter of the function
13. The statement chr(ord('c'))) returns the following
    1. ‘c’
    2. 99
    3. \x0C
    4. ‘C’
    5. 67
14. The built-in function open() uses the following as default newline if none is specified:
    1. \n
    2. \n\r
    3. \r
    4. \d
    5. 0
15. The following mode opens a file to read and write in binary mode:
    1. r+b
    2. rwb
    3. ab
    4. a+b
    5. b
16. The readline() function of a text file object returns lines:
    1. With the trailing newline
    2. Without a trailing newline
    3. With an additional ‘\n’
    4. With an additional ‘\n\r’
    5. With an additional ‘\r’
17. A file object is iterable. This means that
    1. You can request successive members in the object, until the list is exhausted
    2. You can randomly change data in the object
    3. You can determine the size of the object
    4. You can cast the object to another type
    5. The object does not keep data in any order
    6. You can read the second-last byte in a binary file with the following function calls:
18. Given a binary file object f, you can retrieve the last two bytes in the file with the following statement:
    1. f.seek(-1,2)
    2. f.seek(0,-2)
    3. f.seek(-2,0)
    4. f.seek(-2,2)
    5. f.seek(2,-1)
19. The function bytearray([1,2,3]) returns the following type of object:
    1. Bytearray
    2. Byte
    3. Hexadecimal
    4. Binary
    5. String
20. The 7-bit ASCII character set contains the following number of symbols:
    1. 127
    2. 255
    3. 32
    4. 7
    5. 49
21. The statement bytearray(b"\x0A\x0D") returns the following
    1. Bytearray(b’\n\r’)
    2. “\n\r”
    3. ‘0A0D’
    4. Byte(b’\n\r’)
    5. Newline
22. 0b’11110000’|0b’11001100’ equals
    1. 0b‘11111100’
    2. 0b‘00110011’
    3. 0b‘00001111’
    4. 0b‘00111100’
    5. 0b‘10101010’
23. When writing to a binary file, the data must be of the following type
    1. Bytearray
    2. Integer
    3. String
    4. Float
    5. Hexadecimal
24. A string datatype can be written to a file opened in the following mode:
    1. w
    2. wb
    3. r+b
    4. rb+
    5. ab
25. The value represented by b’\xff’ in hexadecimal notation is equal the following binary value:
    1. 11111111
    2. 11110000
    3. 11001100
    4. 10101010
    5. 00001111

## Loops

### The for-loop

One of the most ubiquitous constructs in any computer language is the *for-loop*. The *for-loop* allows a block of code to be repeated a specified number of times. The *for-loop* will have a starting value for a counter, the maximum value for the counter, and the step by which the counter should be incremented every time the loop repeats.

In Python the for-loop is coded as

for x in range(0,10):

Do whatever you want to, keeping all code that is associated with the for-loop incremented

This line is decremented, and is the first line executed after the for-loop

There are a couple of important features that we encounter here for the first time. The **colon** (“**:**”) indicates the start of an **indented block**, and improves the readability of the code. The indented block identifies the **code** that the Python interpreter should execute **before incrementing the loop counter**, and starting from the first indented line of the block of code again. When the counter is incremented to a value that exceeds the maximum value defined in the for…in… line, the interpreter will jump and continue with the line of code directly **following the indented block**.

The *range(start, end, step)* function used in Python returns a **sequence object**. The sequence object is a list of numbers defined by the start, end and step parameters:

print(list(range(0,10,1))) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

The sequence object only contains the start, end and step values, not the entire list of values. It is therefore **very memory efficient**. The *for-loop* iterates over the sequence object, assigning the retrieved value to the variable x, and terminating after the last number in the sequence has been retrieved.

Note that the sequence **includes** the **start**, but **excludes** the **end**. Some more examples:

print(list(range(0,-10,-1)))# [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]  
print(list(range(-10,0,1))) # [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]  
print(list(range(0,10,2))) # [0, 2, 4, 6, 8]

Note that the *for-loop* can iterate over any **iterable** object. We have earlier seen the iteration over a file object:

**for** line **in** my\_file:  
 print(line,end=**''**)

The *for-loop* is typically used to perform the same action repeatedly:

my\_squares = [] # make an 'empty' list object  
**for** x **in** range(0,10): # loop from 0 to 9 (inclusive)  
 my\_squares.append(x\*\*2) # is the index of the current list item x?  
 print(my\_squares[x]) # retrieve the contents from my\_squares[x]  
print(**"done"**)

### Enumerate

It is often useful when you get consecutive entries from a list or any iterable object (see later, in section *Containers and Iterations*), to not simply get the consecutive entries, but also the **index of each entry**. The built-in function *enumerate()* returns a **tuple of (index,value)** when iterating over an **iterable object**:

my\_list = [**'apple'**,**'banana'**,**'pear'**]  
**for** value **in** enumerate(my\_list):  
 print(value) # (0, 'apple') (1, 'banana') (2, 'pear')

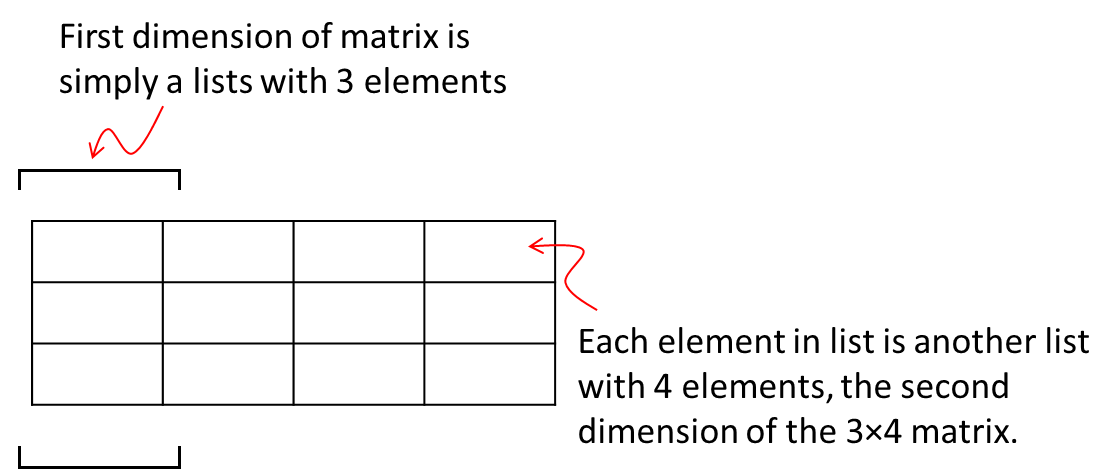
The tuple can also be **directly unpacked** in the same line:

**for** my\_index, my\_value **in** enumerate(my\_list):  
 print(my\_index, my\_value) # 0 apple 1 banana 2 pear

Make a mental note to use **enumerate** when you need **both** the **index** and **item value** in a **for-like loop** construct.

### Nested loops

You can have one loop running within another loop. This is called a **nested loop**. To illustrate the concept we will need a construct that we have not seen before: a **list of lists**. This is, in fact, a **matrix** with *number\_of\_rows* rows and *number\_of\_columns* columns. The “*[0]\*number\_of\_rows*” expression assigns a list with *number\_of\_rows* of elements, all set to *0*, to the variable *my\_matrix*. We then loop over each element of the matrix, and add **another list** of *[0]\*number\_of\_columns* to each element. We end up with a *number\_of\_rows* × *number\_of\_columns* matrix.



**Fig. 12. A two-dimensional matrix can be made with a list, where each list item is another list. Although this is one way to make a matrix, there are more effective ways providing many matrix-specific methods that is part of the Python module *Numpy*.**

number\_of\_rows = 3  
number\_of\_columns = 4  
my\_matrix = [0]\*number\_of\_rows

**for** i **in** range(0,number\_of\_rows):  
 my\_matrix[i] = [0]\*number\_of\_columns  
print(my\_matrix) # [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

Now to illustrate the **nested loop**:

**for** x **in** range(0,number\_of\_rows):  
 **for** y **in** range(0,number\_of\_columns):  
 my\_matrix[x][y] = x\*y  
print(my\_matrix) # [[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]

The **inside loop** is incremented to give each of the columns, and the **outside loop** is incremented to give each of the rows. By running through all the columns for each of the rows in turn, we assign a value to **each cell** in the matrix.

### The while-loop

The *for-loop* executes a fixed number of times until an incrementing “counter” reaches a specified maximum. What if you do not know **how many times** the loop should be executed? Say you need to evaluate an expression every round of the loop **until a condition is not met (becomes False)**? This is where the *while-loop* comes in. The *while-loop* executes repeatedly (to infinity) until a specified condition is not met.

my\_squares = [] # make an 'empty' list object  
x=0  
my\_squares.append(x\*\*2) # x is 0  
**while**(my\_squares[x] <= 1000): # is x\*\*2 smaller or equal to 1000?  
 x+=1 # increment x by 1  
 my\_squares.append(x\*\*2) # append the new x squared to my\_squares  
 print(my\_squares[x]) # print value at current index  
print(**"square larger that 1000!"**) # while loop is done

The condition tested by the while-loop evaluates to either **True** or **False**. In the example above the statement my\_squares[x] <= 1000 or any condition tested must evaluate to **True** for the block of code following the colon to execute. The first time when the condition is not met (i.e., x >1000, and the statement evaluates to **False**) the *while-loop* is **exited**, and the **line of code following the indented while-block** and subsequent lines of code are executed. It is possible to exit a *while-loop* from within the loop (the indented code block) using a break statement, although this is generally considered **bad programming form**. Always make sure that the condition tested does eventually return False, otherwise the *while-loop* will execute as an **infinite** loop. It is also possible (and acceptable) for the condition being tested to return False the very first time it is evaluated, so that the indented code block following the *while-loop* is not executed.

The condition tested can also be a compound statement:

my\_string=**"ABCDEFGHIJKLMNOPQRSTUVWXYZ"**string\_length = len(my\_string)  
x=0  
**while**((x < string\_length) **and** (my\_string[x] != **'Q'**)):  
 print(my\_string[x])  
 x+=1  
print(**'Found it, or not...'**)

Let’s walk through this one. The purpose of the code is to take one character of *my\_string* in turn, compare it to the query character (‘Q’) until a match is found, and making sure that we have not used all characters in the string. The *len()* function is a built-in function that returns the number of elements in an iterable such as a list, string, dictionary and so forth. We are slicing the string (my\_string[x]) to compare each consecutive character to the query character (‘Q’). It we use an index that points past the end of the string (the position *after* ‘Z’), Python will generate an error, which would prematurely terminate the program, a condition that we do not want.

The *while-loop* test whether a condition is True or False. When the condition is a compound statement such as in the example above, Python evaluates the individual conditions from the **left** to the **right**. Since we have a compound ‘and’ condition, it means if either the left-hand or the right-hand condition in False, the entire compound condition is False (refer to Table 7). In the code above Python thus first test if we have an index that is past the end of the string. If we have (condition is False), the while loop **immediately terminates** without evaluating the right-hand condition (it is immaterial since the left-hand condition is False). Thus, we will never try to read a character at an index that is beyond the end of the string, and will therefore never try to evaluate a condition that will generate an error. ***This is an approach that is worthwhile remembering and using in your own code*.**

A difficulty of the present code is that when we get to the line following the *while* block, we do not know whether we have found a match or whether we have exhausted the string. We can test this with a **conditional expression**.

## Conditionals

### If, elif, else

The *while-loop* encountered above will continue executing and looping while a condition evaluates to True. The statement *if* evaluates a condition **once**. If it evaluates to True, the indented code following the *if* statement is executed once. If the condition tested by *if* evaluates to False, the indented code block is **skipped**:

my\_response = int(input(**"Enter a number smaller that 5"**))  
**if**(my\_response < 5):  
 print(**"Well done!"**) # if you followed the instructions...

The condition evaluated by *if* can also be compound:

my\_response = int(input(**"Enter a number between 0 and 5"**))  
**if**((my\_response > 0) **and** (my\_response < 5)):  
 print(**"Well done!"**)

You may be interested in what number was entered. To perform multiple evaluations, you can use *if-elif* constructs. *Elif* is shorthand for *elseif*:

my\_response = int(input(**"Enter a number between 0 and 5"**))  
**if**(my\_response == 1):  
 print(**"you entered 1"**)  
**elif**(my\_response == 2):  
 print(**"you entered 2"**)xxxxxxxxxxxxxxxxxxxxxxxxxxx  
**elif** (my\_response == 3):  
 print(**"you entered 3"**)  
**elif** (my\_response == 4):  
 print(**"you entered 4"**)

The Python interpreter will evaluate the *if* and the *elif* statements until a condition is found that evaluates to True. The interpreter will then execute the associated block of code, and jump to the first line of code following the if-*elif* block. It will skip any additional *elif* statements once one evaluated to True.

It is also possible to use a catchall *else* statement in an *if* block. If the *if* and all *elif* statements evaluate to False, the code following the *else* block is executed. If the *if* or any *elif* statement evaluates to True, the *else* block is skipped.

my\_response = int(input(**"Enter a number between 0 and 5"**))  
**if**(my\_response == 1):  
 print(**"you entered 1"**)  
**elif**(my\_response == 2):  
 print(**"you entered 2"**)  
**elif** (my\_response == 3):  
 print(**"you entered 3"**)  
**elif** (my\_response == 4):  
 print(**"you entered 4"**)  
**else**:  
 print(**"you chose poorly"**)

We can now return to our last example of the previous section, where we were not quite sure if we had found the query character in our alphabet, or whether we exhausted our list of letters. Using an *if-else* statement, we can now test to see which option is True:

my\_string=**"ABCDEFGHIJKLMNOPQRSTUVWXYZ"**string\_length = len(my\_string)  
x=0  
my\_character = str(input(**"Enter one character"**))  
**while**((x < string\_length) **and** (my\_string[x] != my\_character)):  
 x+=1  
**if**(x != string\_length):  
 print(**"found entry at index"**,x)  
**else**:  
 print(**"the query entry was not found"**)

## Exercise Day 3

1. The for x in range(1,10): statement will generate the following sequence of values for x:
   1. 1,2,3,4,5,6,7,8,9
   2. 1,2,3,4,5,6,7,8,9,10
   3. 1,1,1,1,1,1,1,1,1,1
   4. 9,8,7,6,5,4,3,2,1
   5. 10,9,8,7,6,5,4,3,2,1
2. The statement range(1,5,2) stores the following in memory:
   1. The sequence object returned by the range built-in function
   2. 1,2,3,4,5
   3. 1,3
   4. 3,5
   5. 1,3,5
3. The statement for x in range(5,1,-1): will generate the following sequence of values for x:
   1. 5,4,3,2
   2. 5,4,3,2,1
   3. 1,2,3,4,5
   4. 1,2,3,4
   5. -1,-2,-3,-4,-5
4. In the statement for x in object:, object can be replaced by
   1. Any iterable object
   2. Only with a range build-in function
   3. Only with a sequence of values
   4. Only with a list of values
   5. Any non-iterable object
5. A for-loop is typically used to
   1. Repeat a section of code a specified number of times
   2. Repeat a section of code until a condition is True
   3. Repeat a section of code an undefined number of times until a condition is met
   4. Format code into neat blocks
   5. Test the Boolean value of an expression
6. In a nested for-loop, one can
   1. Generate each value of x for each individual value of y
   2. Increment variables x and y independently
   3. Generate values for x defined by the value of y
   4. Generate values for x and y independently
   5. Use only a single variable
7. The statement my\_sequence = [1]\*3 will generate the following for my\_sequence:
   1. [1,1,1]
   2. [1],[1],[1]
   3. [111]
   4. [3,3,3]
   5. [3]
8. Three nested for-loops can typically be used to initialize a
   1. Three-dimensional matrix
   2. A list with three data members
   3. Three independent lists
   4. A two-dimensional matrix
   5. An *n*-dimensional matrix
9. A while loop typically allows repeat execution of a block of code until
   1. A condition is met
   2. A counter matches an upper limit
   3. The iterated object is exhausted
   4. The while expression is True
   5. The block of code is exhausted
10. The following code

i=0

while(i<10 and i%2==0):

print(i)

i += 1

* 1. Will print the integer 0
  2. Will print out even positive integers smaller than 10
  3. Will print out uneven positive integers smaller than 10
  4. Will execute as an infinite loop
  5. Will not print any output

1. A while loop will terminate
   1. The first time the while conditional expression evaluates to False
   2. The last time the while conditional expression evaluates to False
   3. When the iterator is exhausted
   4. The maximum of the range sequence is encountered
   5. When the block is exited with a break statement
2. When a while loop terminates
   1. The line of code following the indented while block is executed
   2. The last line of the indented while block is executed
   3. The last line that was executed before the while block initialized, is executed
   4. The program terminates
   5. User input is required
3. In the compound while statement while (condition 1 and condition 2), the while code block will execute if:
   1. Condition 1 is True and condition 2 is True
   2. Condition 1 is False and condition 2 is True
   3. Condition 1 is True and condition 2 is False
   4. Condition 1 is False and condition 2 is False
   5. Irrespective of the values returned by conditions 1 and 2
4. In the compound while statement while (condition 1 and condition 2), a decision on whether the while block must execute is made:
   1. Directly after evaluating condition 1, if it evaluates to False
   2. Directly after evaluating condition 1, if it evaluates to True
   3. After first evaluating condition 2
   4. After both condition 1 and 2 are evaluated
   5. After evaluating the conditions from the right to the left
5. An if (condition) statement will
   1. Execute the code block once if the condition evaluates to True
   2. Execute the code block once if the condition evaluates to False
   3. Execute the code block repeatedly, incrementing a counter
   4. Execute the code block repeatedly until a set maximum is reached
   5. Immediately execute the line of code following the if code block
6. An if (condition)… else… statement will
   1. Not execute the code block following the else statement, if condition evaluates to True
   2. Will execute the code block following the else statement, if condition evaluates to True
   3. Will execute the code block following the else statement, regardless of the evaluated value of condition
   4. Will execute both code block following if and else, if condition evaluates to True
   5. Will execute both code block following if and else, if condition evaluates to False
7. In an if (condition 1) … elif (condition 2) … elif (condition 3) … block,
   1. Only code block following the first condition that evaluates to True is executed
   2. The code block following each if or elif statement that evaluates to True is executed
   3. If condition 1 evaluates to True, only elif conditions that evaluate to True are executed
   4. If condition 1 evaluates to False, only elif conditions that evaluate to True are executed
   5. Each elif statement can have a local else clause
8. In an if (condition 1) … elif (condition 2) … else block, the code following the else clause will be executed
   1. If neither condition 1 or condition 2 evaluates to True
   2. Will be executed if either condition 1 or condition2 evaluates to False
   3. Will be executed if condition 1 evaluates to True, but condition 2 evaluates to False
   4. Will be executed regardless of the value of condition 1 and condition 2
   5. Will be executed is any elif statement is False
9. An important difference between a while and if statement is
   1. A while block will repeat until a condition is not met
   2. A while block will repeat until a condition is met
   3. An if block will be executed at least once
   4. A while block will be executed at least once
   5. A counter can be incremented in a repetitive if block
10. An important difference between a for and if statement is
    1. A for statement is intended to repeat a block of code
    2. A for statement is intended to evaluate a condition
    3. An if statement is intended to repeat a block of code
    4. For and if statements can be used for the same purpose
    5. An if statement can iterate over a sequence
11. The “:” (colon) and indented lines in a while block
    1. Indicates the scope of the while statement to the interpreter
    2. Is solely to improve readability of the text
    3. Is not required
    4. Can consist of a variable number of spaces
    5. The “:” is not required
12. A while statement that always evaluates to True
    1. Generates an infinite loop
    2. Generates an exception
    3. Is the same as an if statement
    4. Is a code block that executes at least once
    5. Will terminate after the counter exceeds a set maximum value
13. The sequence 20,17,14,11 will be generated by the following statement
    1. for i in range(20,10,-3)
    2. for i in range(10,20,-3
    3. for i in range(20,11,-3)
    4. for i in range(10,19,-3)
    5. for i in range(11,19,3)
14. The correct order for the statements within an if block is
    1. If (condition 1) … elif (condition 2) … else …
    2. If (condition 1) … else … elif (condition 2) …
    3. Elif (condition 2) … if (condition 1) … else …
    4. Elif (condition 2) … else … if (condition 1) …
    5. Else … elif (condition 2) … if (condition 1) …
15. In the compound while statement while (condition 1 or condition 2), a decision on whether the while block must execute is made:
    1. Directly after evaluating condition 1, if it evaluates to True
    2. Directly after evaluating condition 1, if it evaluates to False
    3. After first evaluating condition 2
    4. After both condition 1 and 2 are evaluated
    5. After evaluating the conditions from the right to the left
16. Given my\_list = [‘rock’,’paper’,’scissors’], the statement for a, b in enumerate(my\_list) will return the following values during the iteration:
    1. 0 rock, 1 paper, 2 scissors
    2. (0,rock), (1,paper), (2,scissors)
    3. 0,1,2
    4. rock, paper, scissors
    5. [‘0 rock’,’1 paper’,’2 scissors’]

## Functions

### Simple Functions

We have encountered built-in functions like *print()*, *range()* and *len()* previously. If we have specific **blocks of code** that is useful and that we **use often**, it is very convenient to define these as **functions** that we can **call**, similarly to using print() and other built-in functions.

It is very easy to define a function for a body of code:

**def** function\_name(optional parameters):  
 Indented code to do what you want to do  
 **return**(parameter)

The function name is preceded by the *def* statement to indicate to the interpreter that a function definition follows. The function can receive parameters, listed in the round brackets, followed by the colon, to indicate the **start of the function block**. The function code block is **indented**. The function **need not** return a value. If it does, it uses the *return* statement, followed by the parameters that must be returned in round brackets.

Let us make a simple function that calculates the square of a number:

**def** square(x):  
 x\_squared = x\*\*2  
 **return**(x\_squared)  
  
number = 3  
my\_square = square(number)  
print(my\_square) # 9

***Note***

The function *square()* is **defined before it is used**. This is essential, since the interpreter must know what the function does for the subsequent code to use it.

The function *square* is defined to receive a single value in the variable *x*. The interpreter does not know what the data type of *x* is until a value is passed to the function. The variable *number* is set to 3, and is passed to the function *square*. Note that the value of *number* is copied to the function parameter *x*. Thus *x* is 3 after the function has been called. The square of *x* is calculated in the function and assigned to the variable *x\_squared*. The function returns *x\_squared*. The returned value of *x\_squared* is assigned to the variable *my\_square* in the line of code that called the function, which is then printed.

As you saw for built-in functions earlier, a function can receive more than one parameter:

**def** power(b,e):  
 **return**(b\*\*e)  
  
base = int(input(**"enter a base number"**))  
exponent = int(input(**"enter an exponent number"**))  
print(power(base,exponent))

If a specific parameter often takes the same value, you can define the value as **a default**, and call the function **without** the parameter. In the example below, when the *power* function is called without the exponent parameter, the default value of 2 is used as exponent.

***Note***

Once you have defined a default value, **all subsequent** parameters must also have default values, otherwise the interpreter cannot work out to which parameters you are passing values.

**def** power(b,e=2):  
 **return**(b\*\*e)  
  
base = int(input(**"enter a base number"**))  
exponent = int(input(**"enter an exponent number"**))  
print(power(base,exponent)) # 8  
print(power(base)) # 4

A function can return more than one value. You can use a tuple to accomplish this.

**def** rectangleproperties(l,s):  
 circumference = 2\*l+2\*s  
 area = l\*s  
 **return**((circumference,area))  
  
long = int(input(**"long side"**)) # 3  
short = int(input(**"short side"**)) # 2  
print(rectangleproperties(long,short)) # (10, 6)

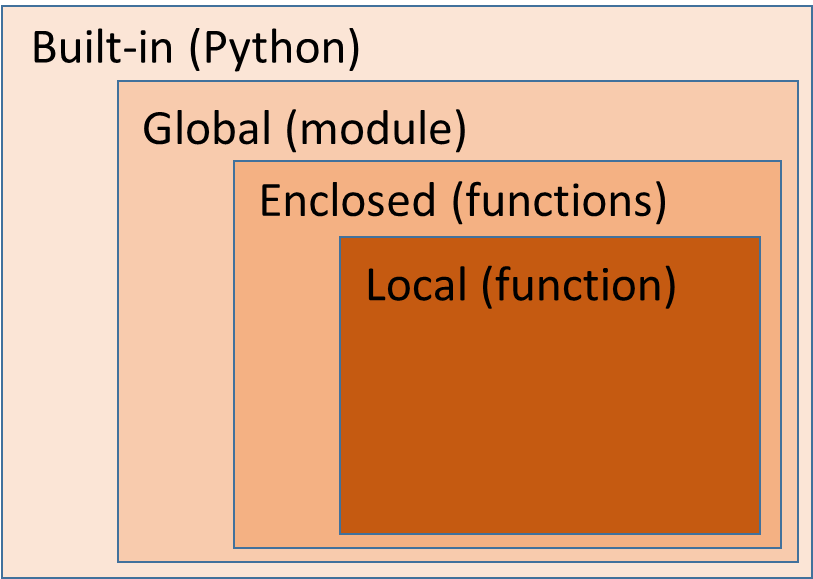
### Variable Scope

Now that we have met **functions**, it is an appropriate time to introduce variable **scope**. Essentially, in Python (and other programming languages), the variable scope is the **neighbourhood where the variable name is known**, also described as **namespace**. **Not knowing** the variable everywhere is very **important**: you may import a module that contains a variable name identical to one in your code. Which variable should the interpreter use? Variable scope makes sure that the interpreter can discriminate between different variables with the same name. For instance:

**def** My\_Function():  
 x = 0  
 print(x) # 0  
  
x = 1  
My\_Function()  
print(x) # 1

Inside the function x is 0; outside the function x is 1. The moment x **is assigned** in the **function**, it becomes a **local variable**, known only in the function. When x is **not assigned** inside the function, it is **global**:

**def** My\_Function():  
 print(x) # 1  
  
x = 1  
My\_Function()  
print(x) # 1



**Fig 13. Variable scope.** The level at which a variable is assigned, determines the scope of the variable as local, enclosed, global, or built-in.

When trying to find the value of a variable, the interpreter will start looking at the level matching that where the variable was encountered, and will progressively step to the broader scopes (outer rectangles in Fig. 13) until an assignment is found. If no assignment is found, the interpreter will generate an error.

NameError: name 'x' is not defined

**def** My\_Function():  
 x=0  
 print(x) # 0  
 **def** My\_Inner\_Function():  
 print(x) # 0  
 My\_Inner\_Function()  
  
x = 1  
My\_Function()  
print(x) # 1

When printing x in My\_Inner\_Function(), the interpreter looks for an assignment to x in My\_Inner\_Function(), does not find one, and then proceeds to look for an assignment in the **enclosing function**, My\_Function(), where an assignment was found, and used. The assignment to the global function (=1) is used at global scope. Note that the term global is ambiguous. In Python, **global** scope really is the **scope of the module** (or file).

When the same variable name is used for a parameter passed to a function, the passed variable becomes a **local variable** because it is assigned **after** the ***def*** keyword.

**def** My\_Function(x):  
 print(x) # 1  
 x=0 # x is a local variable and re-assignment has  
 # no effect of the global x (=1)  
  
x = 1  
My\_Function(x)  
print(x) # 1

When a function returns a parameter, the global variable is only re-assigned if the value returned by the function is assigned to it:

**def** My\_Function(x):  
 print(x) # 1  
 x=0  
 **return**(x)  
  
x = 1  
x = My\_Function(x) # The global variable x is re-assigned (=0)  
print(x) # 0

When you want to assign a value to a global variable inside a function, you can declare the variable global, in which case the assignment is to the global version:

**def** My\_Function():  
 **global** x  
 x=0  
 print(x) # 0  
   
x = 1  
My\_Function()   
print(x) # 0; x was re-assigned in My\_Function()

Make sure you are **comfortable** with the concept of **variable scope**. It can be a nightmare trying to understand why variables with the same name have **different values** in different **namespaces** if you do not keep track of **scope**.

### Recursive functions

A recursive function is a function that **calls itself**. This may sound like a bad idea, because an infinite loop sounds certain. You have to clearly test for a terminating condition to end the recursion. When will you use a recursive function? When you need to repeatedly execute the same bit of code until an end condition is met. Although you can repeatedly call the function from your main body of code, it is tidier to have the function call itself until the terminating condition is met.

**def** power(number, counter):  
 **if**(counter > 0):  
 value = power(number, counter-1) \* number  
 **else**:  
 value = 1  
 **return**(value)  
  
base = 2  
exponent = 4  
print(power(base,exponent)) # 16

Note that the function call within the line of code is like any other operand: the value that is returned by the function is substituted in the place of the function. So in the line of code:

value = power(number, counter-1) \* number

the **value** of “power(number, counter-1)” is substituted in its place. Carefully look at the code of the recursive function. If the counter > 0 the function calls itself with the counter decremented by 1. It carries on calling itself with a decremented counter until the condition counter > 0 is False, i.e., the value of the passed counter is 0. If that happens the function returns the value 1. Where does it return this value? The line of code that called it, which is itself. The easiest way to visualize this is to write the line that causes recursion, and substitute all its parameters with the correct values.

1 # counter = 0

2 = 1 \* 2 # counter = 1

4 = 2 \* 2 # counter = 2

8 = 4 \* 2 # counter = 3

16 = 8 \* 2 # counter = 4

The recursion thus “unwinds” itself, multiplying the result of the previous calculation with 2, passing the result to the function one level up, shown above by the next line down. This is repeated until we get to the original function that started the call with number = 2 and exponent = 4. This is the originating level. The answer is 16. There are of course much easier ways to calculate powers, but the operation lends itself well to illustrating the concept of recursion. Let’s do another example: simple multiplication.

**def** power (number, counter):  
 **if**(counter > 0):  
 value = power(number, counter-1) + number  
 **else**:  
 value = 0  
 **return**(value)  
  
number\_1 = 2  
number\_2 = 4  
print(power(number\_1,number\_2)) # 8

Again, look at the values of the recursive line

value = power(number, counter-1) + number

0 # counter = 0

2 = 0 + 2 # counter = 1

4 = 2 + 2 # counter = 2

6 = 4 + 2 # counter = 3

8 = 6 + 2 # counter = 4

The recursion digs until the terminating condition is met (counter == 0). It then passes the value calculated at this level to the function one level up, and continues passing the calculated result up until the level of the originating function is reached.

## Classes

### Classes

Up to now we have only seen short code snippets. Real programs in bioinformatics are much bigger, and can run into thousands of lines of code. Imagine you want to read the individual chromosome sequences from a fastA file, calculate the AT% of each chromosome, and print it. Not a particularly complex program, but one that will nonetheless take a good number of lines of code. That is, if you program in a linear fashion, starting with the code to read the sequences, storing the sequences in memory, retrieving each sequence in turn, calculating the AT%, and printing the result. Such a type of linear program is old fashioned, and very difficult to maintain, because the code will often jump to positions forwards of backward in the code. The BASIC programming language with its GOTO statement followed by a code line number is an example.

Nowadays programs are object-oriented. Object orientation is a serious topic in computer science, and we must busy ourselves only with what is important. We use object-orientation to make it easier for ourselves to write the code, to make the code easier to understand, and by having the opportunity to re-use bits of code in different programs.

The basic idea behind object-orientation is to group together functions that are relevant to a given data type or to a specific type of calculation, in an object. Remember the functions that we could apply to strings (see Table 6)? Functions such as *capitalize()*, *len()*, *find()* and so forth. These functions were all invoked as my\_string.capitalize(). Here *my\_string* is, in fact, an **object** and the functions in Table 6 all belong to the string object, and can be applied to the object data. However, object functions need not only be applicable to object data. You can code the functions so that they are applicable to any data. For instance, you may want a *Statistics* object, and the *Statistics* object has functions such as *Average()* and *Median()* and so on. So whenever you need to do a statistical calculation, you make your Statistics object, and use its functions:

my\_statistics.median(my\_list)

In this way you keep all the related function logically together in one group. Why am I using *my\_statistics* above, and not *Statistics*? Because we are using an **instantiation (or object),** called *my\_statistics* of the **class** *Statistics*. A class is essentially the description of the **all data types** and **functions** that are available to the object. They are collectively referred to as **attributes** of the object. To continue with pure Python terminology, functions are called **methods** when they are defined as part of a class. In other words, the class described all the methods and data members that it contains. You never use a class directly, You instantiate a class as follows:

my\_instantiation\_of\_class = my\_class()

The *my\_instantiation\_of\_class* is an **instantiation of the class**, or an **object**. **It is the same thing**. So you define a class, instantiate it, and then use the class instantiation object. Let’s look as specific examples:

**class** My\_class:  
  
 **def** Method\_1(self):  
 print(**"This is method 1"**)  
  
 **def** Method\_2(self):  
 print(**"This is method 2"**)  
  
my\_class = My\_class()  
my\_class.Method\_1() # This is method 1  
my\_class.Method\_2() # This is method 2

You instantiate the class *My\_class* with the statement:

My\_class()

The instantiated class or object is then assigned to the variable *my\_class*. You can use the *my\_class* variable (or object) to call the class methods, *Method\_1()* or *Method\_2()*.

You will have noticed that the methods *Method\_1()* and *Method\_2()* were defined with the parameter ***self*** being passed, yet these methods were called from the code with no parameter. What is the story here? The *self* parameter can be thought of as a label that uniquely identifies the object, and is equivalent to the *this* pointer in C++. You can make numerous objects of *My\_class* by repeatedly calling My\_class(), and each would have its own unique *self* identifier. When the interpreter access the object methods, it uses the object identifier *self* “internally” – so you do not have to explicitly supply a value for *self*. Let’s dig some more.

Whenever a class is instantiated, a call to *\_\_init\_\_(self)* is made. This is equivalent to the C++ **constructor**. You can explicitly define this function and initialize some object parameters that you wish to initialize. Similarly, there is a **destructor** *\_\_del\_\_(self)* that is called just before the object goes out of scope (is destroyed).

In the C++ programming language you have data encapsulation where classes have public, protected and private attributes, making specific member functions and data members only accessible to objects or to member functions themselves. **Python has no data encapsulation.** For instance, look at the following code:

**class** Dogs:  
  
 number\_of\_legs = 4  
  
 **def** \_\_init\_\_(self, name):  
 self.dog\_name=name  
  
 **def** \_\_del\_\_(self):  
 **pass  
  
 def** Show\_name(self):  
 print(self.dog\_name)  
  
my\_dog = Dogs(**'Hoppertie'**)  
my\_dog.Show\_name() # Hoppertie  
print(my\_dog.dog\_name) # Hoppertie  
print(my\_dog.number\_of\_legs) # 4

print(Dogs.number\_of\_legs) # 4

We can inspect the data member *dog\_name* using the object method *Show\_name()*, or we can **directly** access it, using the object *my\_dog* as reference as in my\_dog.dog\_name. Class-wide attributes such as *number\_of\_legs* can be accessed via a class reference (Dogs.number\_of\_legs) or the object (my\_dog.number\_of\_legs), although the object reference is a local, object copy, and can independently be manipulated. Be careful, if you change a class attribute value via the class reference it is changed for all instances of that class. If you change it via the object reference, it is only changed for that object:

**class** Dogs:  
  
 number\_of\_legs = 4  
  
 **def** \_\_init\_\_(self, name):  
 self.dog\_name=name  
  
 **def** \_\_del\_\_(self):  
 **pass**my\_dog = Dogs(**'Hoppertie'**)  
my\_dog2 = Dogs(**'Toppeltop'**)  
print(my\_dog.number\_of\_legs) # 4  
Dogs.number\_of\_legs = 6  
my\_dog.number\_of\_legs = 5  
print(my\_dog.number\_of\_legs) # 5  
print(my\_dog2.number\_of\_legs) # 6

### Inheritance

You can extend a class by inheriting it. The base class from which you inherit must be in the same scope as the derived class.

**class** BaseClass:  
  
 **def** \_\_init\_\_(self):  
 self.my\_data = 1  
  
 **def** ShowData(self):  
 **return**(self.my\_data)  
  
**class** DerivedClass(BaseClass):  
  
 **def** \_\_init\_\_(self):  
 self.my\_data = 2  
  
 **def** ShowData(self):  
 **return**(self.my\_data)  
  
my\_base\_object = BaseClass()  
my\_derived\_object = DerivedClass()  
print(my\_base\_object.ShowData()) # 1  
print(my\_derived\_object.ShowData()) # 2

Note that **identically named** object methods and data members **override** any defined in the **base class**. If a method is called that does **not exist** in the **derived** class, the **base** class is **searched** for the specified method. It should therefore be clear that the intention of base class, say ‘Cars’, is to define primitive functions that will be relevant to many different types of objects related to ‘Cars’. In derived classes additional methods and data members can be defined that is of value in a more narrowly specified object such as ‘Electric’ or ‘High\_Performance’ and so forth. Object Oriented Programming (OOP) is a field by itself. If you would like to learn more, view presentations on [YouTube](https://www.youtube.com/results?search_query=object+oeriented+programming+Python+3) or start with books such as [Head First Object-Oriented Analysis and Design by McLaughlin, Pollice and West](https://www.amazon.com/Head-First-Object-Oriented-Analysis-Design/dp/0596008678).

Classes is an excellent tool to develop libraries of functions for yourself from which you can inherit, refine and use in serious bioinformatics applications for years to come.

### Containers and Iterations

We have seen many examples of iteration: in **for-loops**, **list comprehensions** and **generator functions**. Iteration is simply where repeated request to an object gives consecutive data members from a collection until the collection is exhausted. A class, and therefore the instantiated object, can be **iterated** if two specific methods are implemented in the class:

\_\_iter\_\_()

\_\_next\_\_()

Classes that allow iteration over data members are known as **containers**. We have previously met the double underscore \_\_init\_\_() function, which is called when the class is **instantiated**, and allows initiation of any required data members and other housekeeping tasks that must be performed before the object is complete and usable. The **double underscore methods** are **predefined**, and behave in a default manner unless **overridden** by the programmer. If you derive a class from an existing class, all methods defined in the parental class **is inherited**. However, you can define the identical functions in your derived class, and implement different behavior. If the method is called in your derived, instantiated class, your method will be called, not the method with the same name in the parental class. Thus, you can override the \_\_iter\_\_() and \_\_next\_\_() methods.

The data types lists, strings, dictionaries, tuples etc. are all **iterable classes**, implementing \_\_iter\_\_() and \_\_next\_\_(). When Python iterates over an object, it first calls \_\_iter\_\_(). This function should return the reference self, which is interpreted as a reference to **an iterator**. Python then uses the self reference to call the \_\_next\_\_() method repeatedly, which should return consecutive members of a sequence with each call until the sequence is depleted. The following call to \_\_next\_\_() must **raise** the **exception StopIterator**. Exceptions are discussed later. For our discussion here, it is sufficient to know that it is **a system whereby a program raises a notification** when an unusual situation, not generally handled by the program, arises. However, exceptions can be **caught** and **handled**, where a program responds appropriately, and normal program execution continues.

***Note***

Once Python starts calling the \_\_next\_\_() method, it cannot be “reset” and the sequence restarted: the iteration must either be exited (poor style) or completed.

The following class implements iteration over its one data member:

**class** My\_Class:  
  
 # pass a list to the object with initialization**def** \_\_init\_\_(self, list\_data):  
 self.list\_data = list\_data  
 self.index=len(self.list\_data)  
  
 # When Python calls for iteration, set counter to 0**def** \_\_iter\_\_(self):  
 self.counter = 0  
 **return**(self)  
  
 # Check bounds, increment counter, and return list item at previous counter**def** \_\_next\_\_(self):  
 **if** (self.counter < self.index):  
 self.counter += 1  
 **else**:  
 **raise** StopIteration  
 **return** (self.list\_data[self.counter-1])  
  
my\_list = [0,1,2,3,4,5,6,7,8,9]  
my\_object = My\_Class(my\_list)  
**for** i **in** my\_object:  
 print(i,end=**" "**) # 0 1 2 3 4 5 6 7 8 9

**Note**

The \_\_getitem\_\_() method behaves like \_\_next\_\_(), but is **deprecated**, and should not be used.

## Assignment Day 4

#### **Write a program to read the names and sequences from a fastA file, and to calculate the AT% of each sequence**

Download the file “demo\_fasta\_file\_2018.fsa” from SUNlearn, and save it on your computer **in the same directory that will contain your Python program file**. For example, in “C:\Users\hpatterton\PycharmProjects\helloworld\source\demo\_fasta\_file\_2018.fsa”. The exact path will be similar but different on your computer.

Write a program that returns:

1. the number of sequences in the downloaded fastA file, as well as
2. the name of each sequence,
3. the sequence itself, and
4. the AT% of each sequence.

Write the program by defining the following 3 functions that are called in the main code block, below:

Insert your own code to accomplish the intended purpose of each function:

#<<<<<<<<<<<<<<<<<<< COPY CODE BELOW FROM HERE>>>>>>>>>>>>>>>>>>>>

**def** Number\_Of\_Sequences(filepath):

#=====================================================================  
# Return the number of sequences in the downloaded   
# “demo\_fasta\_file\_2018.fsa” file as an integer in the variable  
# ‘number\_of\_sequences’  
#=====================================================================

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
# YOUR CODE GOES HERE  
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
 **return**(number\_of\_sequences)

**def** Read\_FastA\_Names\_And\_Sequences(filepath):

#=====================================================================  
# Return a tuple composed of two lists: ‘sequence\_names’ and ‘sequences’  
# that contain the sequence\_names and sequences read from the downloaded  
# fastA file. The sequence\_names list must contain string elements  
# corresponding to each name, and the sequences list must contain string  
# elements corresponding to each sequence. The string elements may not  
# contain spaces, newline or ‘>’ characters. #=====================================================================

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
# YOUR CODE GOES HERE  
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
 **return**((sequence\_names,sequences))

**def** AT\_Percentage(sequences):

#=====================================================================  
# Return a list in AT\_percentage that contain the AT% of each of the  
# sequences passed in the list ‘sequences’(read from the downloaded  
# fastA file) as a float  
#=====================================================================

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
# YOUR CODE GOES HERE  
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**return**(AT\_percentage)

#=====================================================================  
# MAIN CODE BLOCK  
# This block of code will execute correctly if your functions return  
# the data according to the instructions above  
# **DO NOT MODIFY ANY OF THE CODE BELOW**  
#=====================================================================  
  
filepath=str("demo\_fasta\_file\_2018.fsa")

number\_of\_sequences = Number\_Of\_Sequences(filepath)

print("Number of sequences =",number\_of\_sequences)

sequence\_names,sequences = Read\_FastA\_Names\_And\_Sequences(filepath)

AT\_Percentage = AT\_Percentage(sequences)

for i in range(0,number\_of\_sequences):

print(sequence\_names[i],'\n',sequences[i],'\n',AT\_Percentage[i])

print('done...')

#<<<<<<<<<<<<<<<<<<< COPY CODE ABOVE TO HERE>>>>>>>>>>>>>>>>>>>>

The program should produce output like the following:

Number of sequences = 3

chromosome\_1

GATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGA

50.0

chromosome\_2

GATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATCGATCGAGATC

50.0

chromosome\_3

TTTTGGAAAATTTTGGAAAATTTTGGAAAATTTTGGAAAATTTTGGAAAATTTTGGAAAATTTTGGAAAATTTT

80.0

done…

**Note: your program will be marked by executing it, using a different fastA file to ensure that your code functions properly.**

To submit your program, upload your completed, functioning text file to SUNlearn **using your student number and a “.py” extension as file name**, eg. “2010031648.py”. Your filename must not contain any other characters, spaces or symbols. If your file name format is incorrect, your assignment will not be marked. **Make sure** your file is **functional** and **can execute as a Python script**. If it does not execute correctly, it will not be marked. **Do not e-mail** your program or **submit** it as a **hard copy**. It will not be marked. Programs that are submitted to SUNlearn after the submission deadline will not be marked.

**Please be aware of the policies and rules of Stellenbosch University regarding plagiarism when submitting work as your own.**

MARKS (25)

Number\_Of\_Sequences(filepath) returns the correct number of sequences (5)

Read\_FastA\_Names\_And\_Sequences(filepath) returns a tuple composed of two lists (5)

Read\_FastA\_Names\_And\_Sequences(filepath) returns the correct sequence names in the list 'sequence\_names' (5)

Read\_FastA\_Names\_And\_Sequences(filepath) returns the correct sequences in the list 'sequences' (5)

AT\_Percentage(sequences) returns the correct AT% for each sequence as a list of floats in 'AT\_percentage' (5)

## Modules

### Importing modules

It was previously mentioned that you can write your own classes with methods that are useful for your own work, and then re-use these classes in programs that you subsequently write. If, for instance, you have a class called My\_Class saved in a file called My\_Class\_File.py, you can simply import My\_Class\_File.py as

**import** My\_Class\_File

this loaded file is called a **module**. The paths searched for the specified module is in the following order:

1. The directory of the program file that specifies the import
2. PYTHONPATH (a list of directory names)
3. The installation-dependent default directory

Note that you must omit the ‘.py’ file extension from the name specified for import. You have access to the class or classes specified in My\_Class\_File by **referencing the imported module**:

my\_object = My\_Class\_File.My\_Class()

my\_other\_object = My\_Class\_File.My\_Other\_Class()

Alternatively, you can import a **specific class** as

**from** My\_Class\_File **import** My\_Class

and use it **without reference** as

my\_object = My\_Class()

Apart from classes and functions that you may define and reuse in programs, there are many **third-party modules** that are **extremely useful in Bioinformatics**:

#### **Biopython (**[**biopython.org**](http://biopython.org/)**)**

Biopython includes classes and resources for sequence reading, writing and manipulation, sequence alignment, BLAST, downloading data directly from NCBI and Expasy servers, manipulating and analyzing PDB format data, population genetics and phylogenetic analyses, motif analyses, cluster analyses and supervised learning methods.

Visit the Biopython website and scan the documentation. Many run-of-the-mill tasks in bioinformatics have been addressed and solutions are available in Biopython. Do not re-invent the wheel.

#### **Numpy (**[**www.numpy.org**](http://www.numpy.org/)**)**

Numpy support may fundamental features of scientific programming using Python, including *n*-dimensional arrays, incorporating C/C++ and FORTRAN in Python code, and many useful linear algebra routines, Fourier transforms, and random number functions.

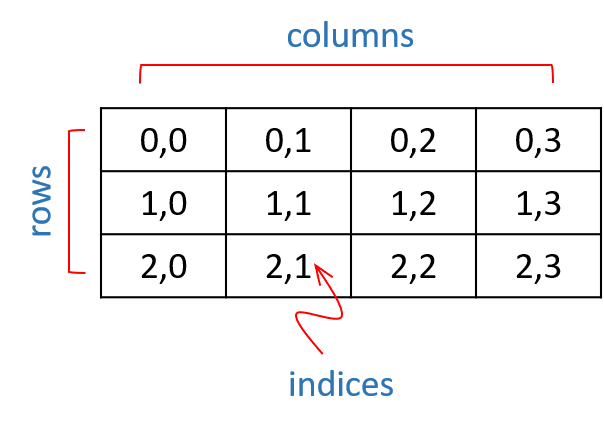
#### **Scipy (**[**www.scipy.org**](http://www.scipy.org/)**)**

Is a core to Numpy and includes many numerical, integration, signal processing, fast Fourier transforms, linear algebra, and graphic processing methods.

### Numpy

We will not be doing a major overview of Numpy in this course. There are many excellent [online tutorials](https://docs.scipy.org/doc/numpy-dev/user/quickstart.html) available that covers the module in depth. However, one important feature in Numpy that we will cover because of its **ubiquitous application in bioinformatics**, is **arrays**. Although the standard Python library **does offer** a functional array data type, its properties and methods are **rather limited**, and users are encouraged to start with using the **ndarray** (for *n*-dimensional array) **class** from Numpy.

An array is a *n*-dimensional list. In Python arrays, a dimension is referred to as an **axis**, and the number of axes, as the **rank**. Two-dimensional arrays or matrices are often used, and is composed of **rows** and **columns**. Each cell in a matrix is addressable by the index of its row and column, both indexed from 0.



**Fig. 14. A 2-dimensional array**

To use ndarray (also known by its alias ‘array’) from Numpy, you must **import the numpy module**:

**import** numpy  
my\_array = numpy.array([0,1,2,3]) # [0 1 2 3]

You can also define an **alias** for the module, allowing a shorthand reference:

**import** numpy **as** np  
my\_array = np.array([0,1,2,3]) # [0 1 2 3]

#### Creating an array

An array is created by passing a **single list** to the class with initialization. If you want a *n*-dimensional array, you can pass a list with the appropriate number of list items:

my\_array = np.array([[0,1,2],[3,4,5]]) # 2-dimensional array

Alternatively, you can pass a 1-dimensional list, and **reshape** the array **after instantiation**. The **number of elements** in the reshaped array must **match** that in the **original list**.

my\_array = np.array([0,1,2,3,4,5])  
my\_array=my\_array.reshape(2,3)

[[0 1 2]   
 [3 4 5]]

my\_array=my\_array.reshape(2,4) # WRONG, 8 items != 6 items

If you do not know the contents of the array at instantiation, you can fill it with 0s, 1s or random floating-point number, defining the **size with a tuple**:

my\_array=np.zeros((2,3))

[[ 0. 0. 0.]   
 [ 0. 0. 0.]]

The default data type is **float64**.

You can define the data type with **dtype**:

my\_array = np.zeros((4, 4),dtype=int)

[[0 0 0 0]

[0 0 0 0]

[0 0 0 0]

[0 0 0 0]]

#### Elementary mathematical operations

Mathematical operations on arrays are usually **elementwise**, i.e. cell A1 *operator* cell B1 = result. Given:

A = np.array([[0,1],[2,3]])  
B = np.array([[1,1],[2,2]])

A+B =

[[1 2]   
 [4 5]]

A-B =

[[-1 0]   
 [ 0 1]]

A\*B =

[[0 1]   
 [4 6]]

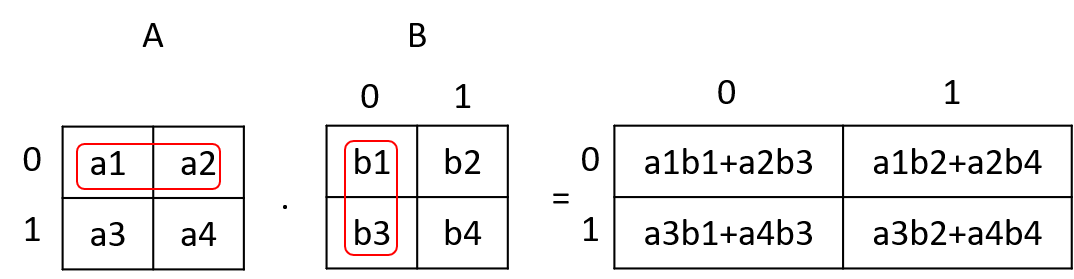
And A/B =

[[ 0. 1. ]   
 [ 1. 1.5]]

The arrays must be of identical size and shape to apply elementary mathematical operands.

#### Dot product

An extremely widely used matrix operation is the **dot product A·B**:



**Fig. 15.** In a dot product, row 0 is applied to column 0, multiplying the parameters is each overlaid cell, and adding the products, and so on, for each row in A and each column in B. The **number of columns** in **array A** must match the **number of rows** in **array B**. It is therefore possible to calculate the dot product of 2×3 and 3×4 matrices, giving a product matrix of size 2×4. In the general case the size is **P×M·M×Q** producing a **P×Q** array.

Given arrays A and B, one can compute the **dot product** using three nested *for-loops* **in a rather cumbersome way**. Note the use of ***arange()***, a ndarray class method similar to the built-in function *range()*, where *arange()* returns an object with data type *numpy.ndarray* with a **1-dimensional list** of **integers** in the **specified range**:

**import** numpy **as** np  
number\_of\_rows\_A = 2  
number\_of\_columns\_A = 2  
number\_of\_rows\_B = 2  
number\_of\_columns\_B = 3  
A = np.arange(number\_of\_rows\_A\*number\_of\_columns\_A).reshape(number\_of\_rows\_A,number\_of\_columns\_A)  
B = np.arange(number\_of\_rows\_B\*number\_of\_columns\_B).reshape(number\_of\_rows\_B,number\_of\_columns\_B)  
C = np.zeros((number\_of\_rows\_A,number\_of\_columns\_B))  
  
**for** row **in** range(0,number\_of\_rows\_A):  
 **for** column **in** range(0,number\_of\_columns\_B):  
 **for** inner\_column **in** range(0,number\_of\_columns\_A):  
 C[row,column] += A[row,inner\_column]\*B[inner\_column,column]

A = [[0 1],[2 3]]   
B =[[0 1 2],[3 4 5]]   
C: [[3. 4. 5.],[9. 14. 19.]]

Alternatively, and **much easier**, the three nested *for-loops* can be replaced with:

C=A.dot(B) # [[3. 4. 5.],[9. 14. 19.]]

Note that the dot product function is **a method of the ndarray class**, and the array object must thus **be referenced**.

#### Sum, min and max

You can also **sum** or calculate the **minimum** and **maximum** or an array:

A = np.arange(6).reshape(2,3)  
print(A.sum()) # 15print(A.min()) # 0print(A.max()) # 5

The axis (dimension) used for sum, min or max can be specified:

print(A) # [[0 1 2],[3 4 5]]print(A.sum(axis=0)) # [3 5 7]print(A.min(axis=1)) # [0 3]  
print(A.max(axis=1)) # [2 5]

#### Indexing and slicing

Individual cells in an array can be accessed with a [row,column] index (or [dimension 1, dimension 2, dimension 3, …] in the case of an *n*-dimensional array):

A = np.arange(6).reshape(2,3)  
print(A[1,2]) # 5

Arrays can also be sliced:

A = np.arange(6).reshape(2,3)  
B = A[0,:] # [0,1,2] row 0, all columns  
C = A[-1,:] # [3 4 5] last row, all columns  
D = A[0:2,0]) # [0 3] rows 0 and 1, column 0

#### Iteration

Arrays can be iterated over. The 0 axis is used:

A = np.arange(6).reshape(2,3)  
**for** item **in** A:  
 print(item)   
 [0 1 2]   
 [3 4 5]

You can also access each item in an array be “flattening” it before iteration:

**for** item **in** A.flat:  
 print(item)   
0  
1  
2  
3  
4  
5

#### Identity and transposed matrices

An **identity matrix (I)** is composed of all zeroes except for the diagonal of ones, and can only be square (N×N). It is very useful in linear algebra because it has the property that

N·I = I·N = N

and in calculating the inverse, A-1, of a matrix A because

A·A-1 = A-1·A= I

An **identify matrix** can be created by specifying the size of one axis:

I = np.eye((3),k=0,dtype=int) # default k=0 and dtype=int  
 [[1 0 0]   
 [0 1 0]   
 [0 0 1]]

The **offset** of the diagonal can be set with *k* to produce a matrix of 0s and 1s that is not the identity matrix:

I = np.eye((3),k=1,dtype=int)   
 [[0 1 0]   
 [0 0 1]   
 [0 0 0]]

Matrices can be **transposed** (**switching the rows and columns**, maintaining their order). Transposed matrices have wide application in linear algebra (and, therefore, bioinformatics, gaming graphics, signal and image processing, etc.):

A = np.array([0,1,2,3]).reshape(2,2)  
A\_t = A.transpose()  
print(A) # [[0 1] [2 3]]print(A\_t) # [[0 2] [1 3]]

A matrix need **not be square** to be **transposable**:

A = np.array([0,1,2,3,4,5]).reshape(2,3)  
A\_t = A.transpose()  
print(A) # [[0 1 2] [3 4 5]]print(A\_t) # [[0 3] [1 4] [2 5]]

## Exceptions

Properly coded programs generally executes successfully under most conditions, terminating with exit code 0. However, even properly coded programs may encounter unexpected conditions. For instance, the file that the program tries to read may be in an unexpected format, or the denominator of a fraction has been rounded to zero, or the input from a user exceeds a prescribed range. It is **impossible** when coding a program to think of and preempt **every possible problem** that a program may encounter. This is why Python and other modern computer languages allow **exceptions**.

When a condition arises that cannot be handled by the code that is being executed, an exception is **raised** or **thrown**.

print(x/0) # ZeroDivisionError: division by zero

Typically, the interpreter will report the exception, and then terminate the program, reporting an exit code of 1. However, it is possible to **catch an exception**, **handle it correctly**, and then either exit the program gracefully, or continue executing the program. An exception is caught in a *try…except* block. As a simple example, this code generates an exception when the calculation of log10 of 0 is attempted:

**import** math  
my\_list = [1000,100,10,0]  
**for** x **in** my\_list:  
 print(math.log10(int(x))) # 3.0 2.0 1.0 ValueError: math domain error

We can catch the exception with a try block:

**for** x **in** my\_list:  
 **try**:  
 print(math.log10(int(x)))  
 **except** ValueError:  
 print(**'not defined'**)

The statement in the try block is executed. If no exception is raised, the interpreter continues with the line after print('not defined'). If an **exception is raised**, an **except block that matches the error** is searched for. If one is found (an attempt to calculate log10 of 0 raises a ValueError exception), the block of code in the **exception block** is **executed**.

3.0

2.0

1.0

not defined

The program then continues with the **first line following** the *try…except* block. It is possible having multiple *except* statements. Note that the **first matching** *except* block is executed.

**try**:  
 print(math.log10(int(x)))   
**except** ZeroDivisionError:  
 print(**'division by 0 is undefined'**)  
**except** ValueError:  
 print(**'not defined'**)  
**except** NameError:  
 print(**'not a number'**)

3.0

2.0

1.0

not defined

Exception values can also be tested as a **tuple**:

**try**:  
 print(math.log10(int(x)))  
**except** (ZeroDivisionError,ValueError,NameError):

print('one of 3 possible errors')

a **catch-all** exception does not define a error value:

**try**:  
 print(1000/x)  
 print(math.log10(int(x)))  
**except**:  
 print(**'this can be any exception...'**)

Note: simply catching and ignoring all exceptions (as we do above) is not a good idea, because the program may continue executing with a value assigned to a variable that will cause trouble later, forcing us to terminate the program. Catch and deal with exceptions as quickly as possible.

When an exception has been caught in an except statement, but we cannot deal with it, it can always be **re-raised**, in which case it will be caught by the **default exception framework**:

**try**:  
 print(math.log10(int(x)))  
**except**:  
 print(**'this can be any exception...'**)  
 **raise**

3.0

2.0

1.0

this can be any exception...

Traceback (most recent call last):

File "C:/Users/hpatterton/PycharmProjects/Test\_main/src/test\_main.py", line 7, in <module>

print(math.log10(int(x)))

ValueError: math domain error

Process finished with exit code 1

If you want to execute a body of code if an exception is **not raised**, you can use an *else* statement following the except statement or statements:

**try**:  
 print(math.log10(int(x)))  
**except** ValueError:  
 print(**'math domain error'**)  
**else**:  
 print(**'OK'**)

3.0

OK

2.0

OK

1.0

OK

math domain error

It is also possible to execute a body of code **regardless** of whether an exception has been thrown or not, using the *finally* statement:

**try**:  
 print(math.log10(int(x)))  
**except** ValueError:  
 print(**'math domain error'**)  
**else**:  
 print(**'OK'**)  
**finally**:  
 print(**'Another round'**)

The code in the finally block is **always executed**. The standard exceptions are listed in Table 15.

**Table 15.** Standard exceptions

|  |  |
| --- | --- |
| Exception Name | Description |
| Exception | Base class for all exceptions |
| StopIteration | Raised when the next() method of an iterator does not point to any object. |
| SystemExit | Raised by the sys.exit() function. |
| StandardError | Base class for all built-in exceptions except StopIteration and SystemExit. |
| ArithmeticError | Base class for all errors that occur for numeric calculation. |
| OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| FloatingPointError | Raised when a floating point calculation fails. |
| ZeroDivisionError | Raised when division or modulo by zero takes place for all numeric types. |
| AssertionError | Raised in case of failure of the Assert statement. |
| AttributeError | Raised in case of failure of attribute reference or assignment. |
| EOFError | Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| ImportError | Raised when an import statement fails. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| LookupErro | Base class for all lookup errors. |
| IndexError | Raised when an index is not found in a sequence. |
| KeyError | Raised when the specified key is not found in the dictionary. |
| NameError | Raised when an identifier is not found in the local or global namespace. |
| UnboundLocalError | Raised when trying to access a local variable in a function or method but no value has been assigned to it. |
| EnvironmentError | Base class for all exceptions that occur outside the Python environment. |
| IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| OSError | Raised for operating system-related errors. |
| SyntaxError | Raised when there is an error in Python syntax |
| IndentationError | Raised when indentation is not specified properly. |
| SystemError | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| SystemExit | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| ValueError | Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| RuntimeError | Raised when a generated error does not fall into any category. |
| NotImplementedError | Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |

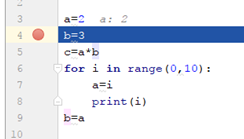
## Debugging

Debugging is not an afterthought effort to find errors in a program introduced by sloppy programming, but rather a consistent and rigorous approach to **ensure** that a **program executes as intended**. This is particularly crucial in scientific research. You must convince yourself that the **output** produced by the program is **accurate and correct** under as many testable conditions as possible. This is typically achieved by testing and debugging smaller sections of the program as you finish coding them. *Get into the habit of debugging every function or conceptual code block in a program as you finish them.* You will typically test a function with a **test data set** that will produce **predictable output**, and ensure that the function **indeed does produce** the **expected output**. Debugging will also ensure that functions or program sections can **handle the size of datasets** used, that calculation proceeds properly at **data extremities**. For instance, if you calculate values in a scanning window, is the scanning window initially aligned with the start of the data set, as well as with the end of the data set at the final setting of the window? How does your program behave if the format of the data input file is incorrect? What does the program do if the size of the dataset that it tries to read exceeds the memory of the computer? These are all questions that you must answer before using the program in a serious scientific application or making the program available to a wider bioinformatics community.

If, during debugging a program, you find a section of the program or a function that does not behave as expected or as you intended, you must carefully analyze the function or program section to **understand the basis** of the **unexpected behavior**, and to **fix** it.

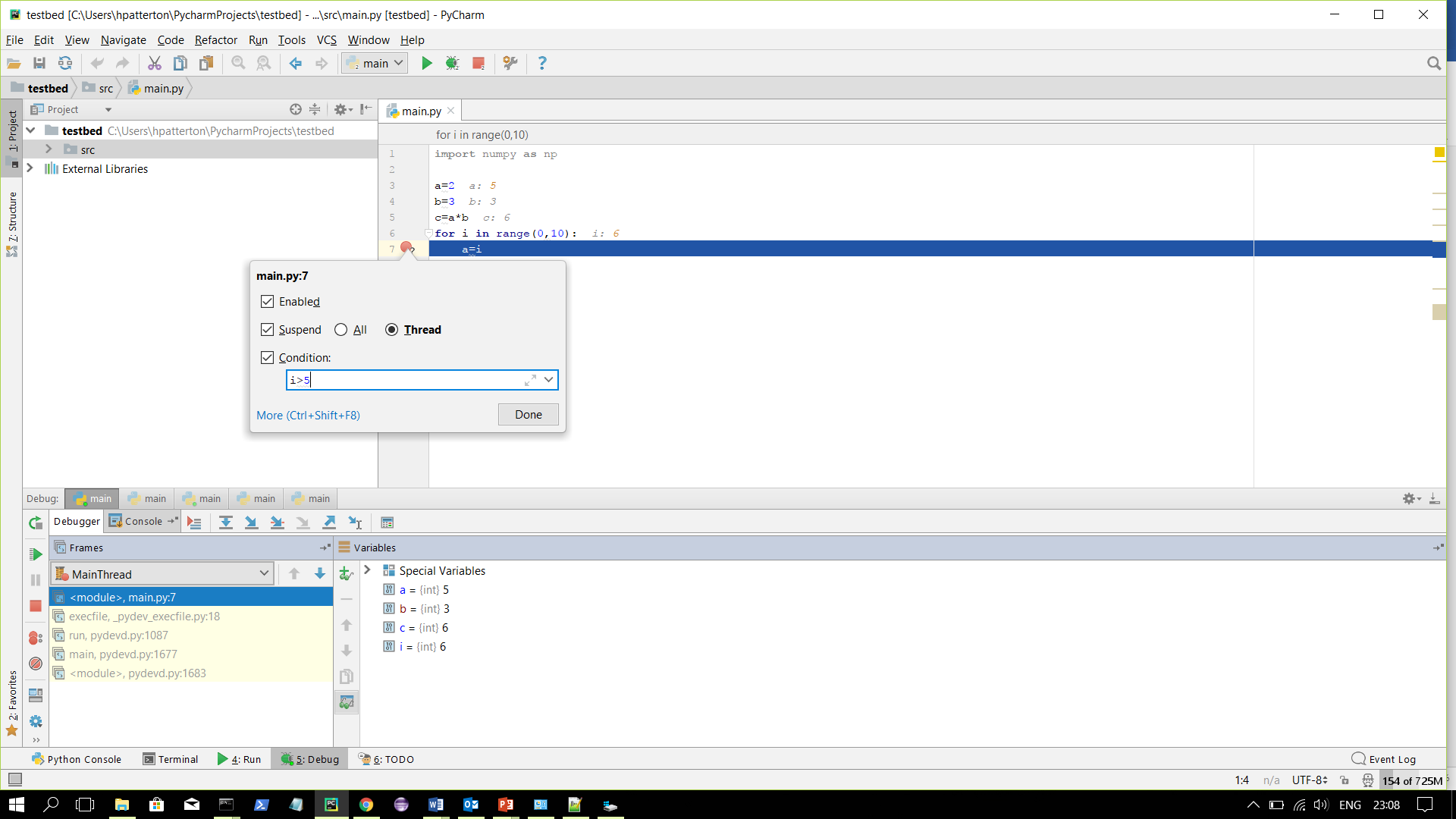
### Breakpoints and conditional breakpoints

One of the first things to do when debugging a program, is to **pause the execution** of the program at a given point to inspect the values of variables. This is accomplished by setting a **breakpoint**. This is done by **double clicking** in the gutter margin next to the line where you want the program to pause. A red circle icon will appear. Clicking on the icon cancels the breakpoint. When you now select **Run | Debug (Shift+F9)** from the menu, the interpreter will interpret the program from the first line of the program **up to the line** with the **breakpoint** set. The values of all **variables** that exist within the scope of the breakpoint will be **displayed** in a Variable window.



***Fig. 16. Setting a breakpoint***

When setting a breakpoint, you can also right click on the red circle icon and **define a condition** that must evaluate to True for the breakpoint to pause the program.



***Fig. 17. Defining a conditional breakpoint.***

You can set several breakpoints in a program. When the program is paused at a breakpoint, you can allow the program to continue running by selecting **Run | Resume Program (F9)** from the menu. You can also click on a line to place the cursor on that line, and then select **Run | Run to Cursor (Alt+F9)** from the menu to allow the program to run **up to the line** that you selected.

### Single stepping

If you want to execute the program **line by line** to view the values of variables, you can select single stepping, **Run | Step Over (F8)** to interpret consecutive single lines of code. Note that when the interpreter reaches a function, Step Over will allow the interpreter to **execute the contents of the function as a block**, and then **pause** at the **line after** **the function**. If you are interested in single stepping **into** the **code in functions**, choose **Run | Step Into (F7)**. Step Into can also be used to single step lines outside of functions. It simply dives into function code when it is encountered, whereas Step Over skips the display of the code.

Note that you can do most of your debugging using the F7, F8 and F9 keys (Windows OS). Memorize these: it makes the debugging process much more efficient than continuously visiting the menu.

## Assignment Day 5

#### **Write a program to read the names and sequences from a fastA file, and to read the start, stop and translation offset positions of all coding sequences from a gff file, and use the information from the gff file to retrieve the sequence sections and determine the frequencies of all 64 possible codons in the yeast coding sequences.**

First, download the files “saccharomyces\_cerevisiae\_2018.fna” and “saccharomyces\_cerevisiae\_2018.gff” from SUNlearn, and save it on your computer **in the same directory that will contain your Python program file**. For example, in “C:\Users\hpatterton\PycharmProjects\helloworld\source\saccharomyces\_cerevisiae\_2018.fna”. The exact path will be similar, but different, on your computer.

In today’s assignment you will need to define the following three classes and associated class methods:

|  |  |  |
| --- | --- | --- |
|  | Class | Method |
| 1 | Read\_FastA | Read\_FastA\_Names\_And\_Sequences(self, filepath) |
| 2 | Read\_GFF | Get\_Gene\_Positions(self, list\_of\_gff\_features, filepath, feature) |
| 3 | My\_Codons | Count\_Codons(self, sequence, codons, number\_of\_occurrences, offset=0) |

It is not necessary to initialize any data members in any class \_\_init\_\_() method.

The Read\_FastA\_Names\_And\_Sequences(self, filepath) method is identical to the function you coded earlier. Simply define the class Read\_FastA and add your function as a class method, adding the ‘self’ reference. Alternatively, if your function from earlier did not work properly, you can use the supplied code.

The function should read the names and sequences of the 17 chromosomes of *Saccharomyces cerevisiae* from the file ‘saccharomyces\_cerevisiae\_2018.fna’ and return a tuple of two lists sequence\_names and sequences, each with 17 string items corresponding to the names and sequences of the 17 chromosomes.

For class ‘Read\_GFF’, you need to code the function ‘Get\_Gene\_Positions(self, list\_of\_gff\_features, filepath, feature)’. **Refer to the Appendix to understand the format of a GFF file.** Unlike earlier, where you simply stepped though the fastA file line-by-line, testing whether a line started with ‘#’ and then using the whole line if it qualified, the **processing** you need to do with the GFF file is a bit more involved:

You will need to step though the GFF text line-by-line, filtering and selecting lines that contain information (does **not** start with the ‘#’ character). You then need to take the usable lines and split it into the 9 items, corresponding to the 9 columns, so that you can work with the data from any single column. You must then retrieve the item from column 2 (index starts at 0) and test whether it equates to ‘CDS’. If it does, you must use the line. If it does not, you must discard the line.

For instance, the first line in the GFF file (open in Notepad++ or Wordpad) that lists a ‘CDS’ feature is:

chrI SGD CDS 335 649 . + 0 Parent=YAL069W\_mRNA;Name=YAL069W\_CDS;orf\_classification=Dubious

Thus, find the lines in the GFF file where column 2 equates to ‘CDS’, and copy the data for **seqID**, **start**, **end** and **offset** (columns 0, 3, 4 and 7 [blue items]) of the line to a tuple, (chrI,335,649,0), and append that **tuple** to the **list** ‘list\_of\_gff\_features’. You do the same for each line where column 2 equates to ‘CDS’. You should end up with a list of approximately 6000 tuples in list\_of\_gff\_features.

***TIP***

An easy way to split a line into a list composed of the nine items separated by tab characters, is to use the *split*() command list\_of\_column\_items = gff\_line.split(sep='\t'). You can then generate and add the tuple to the list\_of\_gff\_features list with the statement list\_of\_gff\_features.append(list\_of\_column\_items[0], list\_of\_column\_items[3], list\_of\_column\_items[4], list\_of\_column\_items[7]).

The Get\_Gene\_Positions function also receives the variables filepath and feature, the path to the ‘*saccharomyces\_cerevisiae\_2018.gff’* file, and the string ‘*CDS*’. We pass ‘CDS’ as a string to the class method in variable feature, so that we can also read lines where the *feature* equates to *gene* or *mRNA* etc., although we will not use these features in this assignment. The function must **return** the list list\_of\_gff\_features with tuples of CDS information as described above.

Next, the My\_Codons class must contain the function Count\_Codons(self, sequence, codons, number\_of\_occurrences, offset=0). This function receives a single DNA sequence from the main body of code (see below), the list of codons generated by the method Make\_List\_Of\_Codons(self), the list number\_of\_occurrences, which is a **list of 64 integers**, and offset, which is the offset (gff file column 7) of the CDS. When the program starts, all 64 entries in Number\_of\_occurrences equals 0. In Count\_Codons the number of occurrences of each of the 64 codons are counted in the sequence passed to the method, and these numbers are **added to** the matching codon entries of number\_of\_occurrences.

***TIP***

The code in your function should contain a statement similar to:

Number\_of\_occurrences += codon\_count\_for\_this\_sequence,

where codon\_count\_for\_this\_sequence is also a list of 64 integers, corresponding to the number of times that each of the 64 codons were counted in the sequence passed to Count\_Codons(). Number\_of\_occurrences is the cumulative value that will finally contain the tally for all 64 codons counted in **all sequences**.

Be careful when you count codons! You cannot simply use the string.count(query) function, since it will count **all non-overlapping** query strings, including the ones **out of frame**. You want to start at the start+offset defined for the CDS (offset is usually 0), and then count in **steps of 3** nucleotides to remain **in-frame** for the CDS.

***TIP***

It is likely that you will use a statement similar to:

for start in range(offset,len(sequence),3):

#<<<<<<<<<<<<<<<<<<< COPY CODE BELOW FROM HERE>>>>>>>>>>>>>>>>>>>>

class Read\_FastA:

#==================================================================

# This function return a tuple of 2 lists 'sequence\_names' and

#'sequences' containing the sequence names and the sequences of the

# entries in the supplied fastA file. There should be no spaces,

# newlines or '>' characters in either 'sequence\_names' or

# 'sequences'. Sequence\_names must be a list composed of string

# items corresponding to each sequence name sequences must be a

# list of string items, each item corresponding to one sequence

#

# YOU CAN USE YOUR CODE FROM DAY 4, OR THE CODE BELOW

#

#==================================================================

def Read\_FastA\_Names\_And\_Sequences(self, filepath):

print("Reading fastA sequences...")

self.sequence\_names = []

self.sequences = []

f = open(filepath, 'r')

self.counter = 0

for i in f:

if (i[0] == '>'):

self.counter += 1

self.sequence\_names.append(i[1:].replace('\n', ''))

self.sequences.append(str())

else:

self.sequences[self.counter - 1] = self.sequences[self.counter - 1] + i.replace('\n', '')

f.close()

return (self.sequence\_names, self.sequences)

class Read\_GFF:

def Get\_Gene\_Positions(self, list\_of\_gff\_features, filepath, feature):

#==================================================================

# This function should be passed a list 'list\_of\_gff\_features' to

# which you append a tuple (seqID, start, end, offset) that

# contains the information from each line from the GFF file

# corresponding to a coding sequences (CDS). Thus, the information

# from the line

# 'chrI SGD CDS 335 649 . + 0

# Parent=YAL069W\_mRNA;Name=YAL069W\_CDS;orf\_classificat...'

# must be appended to the list\_of\_gff\_features as

# ('chrI','335','649','0'). Step through each line of the GFF

# file, selecting only the ones with column 2 == 'CDS', and append

# the seqID, start, end, offset information of each as a tuple to

# 'list\_of\_gff\_features'. Filepath is the full path to the

# 'saccharomyces\_cerevisiae\_2018.gff' file and feature == 'CDS'

#==================================================================

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# YOUR CODE GOES HERE

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

return(list\_of\_gff\_features)

class My\_Codons:

#==================================================================

# The function generates a list of all possible combinations of 3

# of the 4 nucleotides G, A, T and C, and returns the list as

# 'codons'

#==================================================================

def Make\_List\_Of\_Codons(self):

self.nucleotides = ['G','A','T','C']

self.codons = []

self.tempcodon = ''

for a in self.nucleotides:

for b in self.nucleotides:

for c in self.nucleotides:

self.tempcodon = a+b+c

self.codons.append(self.tempcodon)

return(self.codons)

def Count\_Codons(self, sequence, codons, number\_of\_occurrences, offset=0):

#==================================================================

# The string.count(substring,start,end) method may look like a

# suitable function to use, but it finds the occurrence of a

# substring in ANY FRAME. Codons are arranged in non-overlapping

# groups of three. So we have to begin searching groups of three,

# starting at the beginning of the sequence, taking care of any

# offset defined in the gff file, and then jumping by three bases

# after each comparison. We write our own function to do exactly

# this.

#==================================================================

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# YOUR CODE GOES HERE

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

return(number\_of\_occurrences)

#==================================================================

#

# MAIN CODE

#

# This block of code will execute correctly if your functions return

# the data according to the instructions above

#

# **DO NOT MODIFY THE CODE BELOW EXCEPT THE FILE PATHS**

#

#==================================================================

path\_of\_gff\_file = 'saccharomyces\_cerevisiae\_2018.gff' # change the path string if yours is different

path\_of\_fasta\_file = 'saccharomyces\_cerevisiae\_2018.fna' # change the path string if yours is different

# Get the positions and offset of all codings sequences (CDS) in the yeast genome

GFF\_file\_object = Read\_GFF()

list\_of\_gff\_features=[]

total\_sequence\_length = 0

list\_of\_gff\_features = GFF\_file\_object.Get\_Gene\_Positions(list\_of\_gff\_features, path\_of\_gff\_file,'CDS')

# make a list of all 64 possible codons

codon\_object = My\_Codons()

codons = codon\_object.Make\_List\_Of\_Codons()

# Read the chromosome sequences

FASTA\_file\_object = Read\_FastA()

sequence\_name, sequences = FASTA\_file\_object.Read\_FastA\_Names\_And\_Sequences(path\_of\_fasta\_file)

# Loop over list\_of\_gff\_features, using one entry at a time

number\_of\_occurrences =[0]\*64

print('Counting codons...')

for gff\_line in list\_of\_gff\_features:

# get chromosome and slice the gene sequence of the chromosome with the calculated index

chromosome\_sequence = sequences[sequence\_name.index(gff\_line[0])]

number\_of\_occurrences = codon\_object.Count\_Codons(chromosome\_sequence[int(gff\_line[1])-1:int(gff\_line[2])], codons, number\_of\_occurrences, int(gff\_line[3]))

# Print out the total codons

total\_codons = sum(number\_of\_occurrences)

for i in range(0,64):

if(i == 0):

print('Codon','Number','Frequency (/1000)')

print(codons[i],number\_of\_occurrences[i],1000\*number\_of\_occurrences[i]/total\_codons)

#<<<<<<<<<<<<<<<<<<< COPY CODE ABOVE TO HERE>>>>>>>>>>>>>>>>>>>>

Your program output should be similar to the following:

Reading fastA sequences...

Counting codons...

Codon Number Frequency (/1000)

GGG 19582 6.481260214885782

GGA 38383 12.70402465672357

GGT 52540 17.389715641410426

.

.

.

CCA 43695 14.462193090053837

CCT 35146 11.63264076766294

CCC 19864 6.5745967168058

**Note: your program will be marked by executing it, using different fastA and gff files to ensure that your code functions properly.**

To submit your program, upload your completed, functioning text file to SUNlearn **using your student number and a “.py” extention as filename**, eg. “2010031648.py”. Your filename must not contain any other characters, spaces or symbols. If your filename format is incorrect, your assignment will not be marked. **Make sure** your file is **functional** and **can execute as a Python script**. If it does not execute correctly, it will not be marked. **Do not e-mail** your program or **submit** it as a **hard copy**. It will not be marked. Programs that are submitted to SUNlearn after the submission deadline will not be marked.

**Please be aware of the policies and rules of Stellenbosch University regarding plagiarism when submitting work as your own.**

MARKS (25)

Classes and methods are correctly defined (5)

Get\_Gene\_Positions(list\_of\_gff\_features, filepath, feature) returns correct tuples composed of seqID, start, stop and offset. (10)

Count\_Codons(sequence, codons, number\_of\_occurrences, offset) return correct cumulative number\_of\_occurrences corresponding to the number of each of the 64 in-frame codons in the test data (10)

# Appendix

## File formats

### fastA

A fastA format file is a text file with a title line for each sequence in the file. The file can contain a single sequence, or multiple sequences. In the latter case each individual sequence is preceded by a title line. The length of each line is not specified, but is usually 80 characters wide (excluding newline characters). Each title line is started with a “>” character. There is no limit on what characters to use in the title line, except that the “>” is reserved as the first character. Using “>” more than once may confuse some parsers.

>sequence\_1

GATCGATCGATCGACTGA

>sequence\_2

AATTAATTAATTAATTAATT

***Note***

Each line is terminated with a **newline character**. On Windows it is represented by the two escapes characters \n\r and on Linux by a single \n. Be aware of this difference.

See [en.wikipedia.org/wiki/FASTA\_format](https://en.wikipedia.org/wiki/FASTA_format).

### fastQ

This is a text based file of sequences with associated scores for the quality of each nucleotide. It is typically generated with a sequencing apparatus such as manufactures by Illumina or Ion Torrent. The file is composed of units of four lines:

Line 1: starts with the “@” character followed by the name or ID of the sequence

Line 2: the nucleotide sequence

Line 3: starts with a “+” character and may be followed by the sequence name

Line 4: the single letter quality score associated with each nucleotide in the sequence.

An example:

@HWI-ST193:439:D16G8ACXX:4:1101:6038:2128 1:N:0:ATCACG

AGCAGCAATCAGAGATGAAGCCAATGGTGGTCCACGAGCTCCAAATCCTA

+

CCCFFFFFHHHHHJJJJJJJJJJJJJJHHJGIJJJJIJJJJJJIIJJJJI

The sequence name or ID on the first line of the block may also contain additional information. The Illumina sequencing program passes information about the flow cell and sequencing spot position and so forth on this line.

***Note:*** the sequence quality score includes the “@” character, so this character may appear as the first character on line 4. Thus, when parsing, you cannot simply look for the start of the block using the “@” character. You may want to use “@”, skip one line, and then a “+” to positively identify each block of 4 lines.

See: [en.wikipedia.org/wiki/FASTQ\_format](https://en.wikipedia.org/wiki/FASTQ_format)

### Generic Feature Format version 3 (GFF3)

GFF3 format is a text file that lists the location of specific features such as genes or exons or replication origins relative to a defined landmark, usually a chromosome. The GFF3 format is widely used by Genome Browsers to display genes, exons etc. as tracks. There are several versions of GFF files with different specifications. Make sure of the version of GFF file that you work with.

GFF3 format text files **must start** with

##gff-version 3

and may be followed by no or a variable number of comment lines, all starting with a #.

The body of the GFF3 file is composed of **9 tab-separated columns**:

***Note***

*In GFF3 format files, characters are escaped with the “%” character, followed by the hexadecimal value of the escaped character. Thus, tab is “%09” (not \t), newline “%0A” (not \n) and carriage return “%0D” (not \r). Some users, such as SGD, does not seem to comply with this specification.*

Where information in a column is not available, the entry must be “.”; it **cannot be left empty**.

1. ID

The ID or name of the feature.

1. Source

The operating procedure that generated this feature, eg. "Genescan" or a database name, such as "Genbank", etc.

1. Type

The type of the feature such as “gene”, “CDS”, “exon”, etc. This is constrained to be either a term from the Sequence Ontology or an SO accession number.

1. Start

The start position of the feature relative to the landmark given in column 1. The landmark position starts at **index 1 not index 0**. Start is always less than or equal to end.

1. End

The end position of the feature relative to the landmark given in column 1. The landmark position starts at **index 1 not index 0**. End is always larger than or equal to Start.

1. Score

The score of the feature (sequence similarity, probability; a score defined by the user). A floating point number.

1. Strand

The strand of the feature. + for positive strand (relative to the landmark), - for minus strand.

1. Phase

Indicates whether the codon start at phase 0,1 or 2 relative to the start of the feature.

1. Attributes

A list of feature attributes.

See: [github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md](https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md) for the full specifications