Babraham Bioinformatics

Advanced Python

Further Python programming concepts

Version 2020-08

Licence

Table of Contents

Introduction

What this course covers

This course is an immediate continuation of the topics laid out in the Babraham Bioinformatics Introduction to Python syllabus. While it is not essential that you have taken part in that course, you will nevertheless need to have at least working knowledge of the concepts covered in that primer on Python programming.

This advanced course builds on the knowledge covered in that introductory course and it is assumed participants are familiar with basic Python concepts, as listed below.

Assumed prior knowledge

- how to write and run Python scripts (using Thonny)
- Python datatypes
- names (variables)
- functions and methods
- collections (ranges, tuples, lists, dictionaries)
- conditionals
- loops and iterations
- reading from and writing to files

From this starting point, this advanced Python course covers the topics listed below.

Advance Python course overview

- writing better-structured and more robust code
- introducing object-oriented programming
- using Python modules
- pattern-recognition with Regular Expressions

Having successfully completed the lectures and exercises, participants should consider themselves able programmers, with a broad understanding of Python concepts. And what is more, participants should be in a position to tackle more complex, real-world challenges commonly encountered in the research arena.

Chapter 1: writing better code

Comprehensions – succinct alternatives to conditionals and loops

In the Introduction to Python course we met conditional expressions, loops and the closely related iterations. When used together these key components of the Python language prove extremely versatile when it comes to modifying data. In addition, Python has an elegant way achieving these same tasks but in single lines of code in what are known as **comprehensions**. These comprehensions can be used to create sets, lists or dictionaries from using a collection as a starting point. It is worth taking the time to become familiar with the syntax of comprehensions, since by using them you should be able to write more succinct elegant code.

List Comprehensions

List comprehensions generate **lists as output** and have the following template syntax:

```
[expression for item in collection]
```

That is to say, this comprehension will perform the specified expression on every item in the collection. To illustrate this point, look at the next example in which we generate square numbers using 0 to 10 as the root values. Sure, we could use a loop to achieve the same task, but a one-line list comprehension is a succinct Pythonic way to manipulate collections.

```
S = [x**2 for x in range(11)]
print(S)
```

```
>>>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

We created a collection object with the term `range(11)`, and we iterated over this range, assigning each element to `x`. Each value of `x` is subsequently squared and added to the list generated by the comprehension. This list is then assigned to the name `S`.

Set Comprehensions

Set comprehensions are similar to list comprehensions and have a similar syntax, but are enclosed in curly brackets rather than round brackets. As their name suggests, they generate **sets** as output.

```
{expression for item in collection}
```

It is worth bearing in mind that sets only contain unique items, and so you cannot be sure that the output will have the same number of elements as the input. We see in the following example a list containing duplicates of the letter ‘C’. Using the set comprehension, every value from the list (assigned the name `x`) is allocated to the newly generated set. The set has no duplicate values.

```
my_list = ['A', 'B', 'C', 'C', 'C']
my_set = {x for x in my_list}
print(my_set)
```

```
print(type(my_set))


>>>
{'B', 'C', 'A'}
<class 'set'>
```

Dictionary Comprehensions

Essentially the same rules apply to **dictionary comprehensions** as for set comprehensions, but of course dictionaries are produced as output. Dictionary comprehensions have the template:

```
{key-expression: value-expression for key, value in collection}
```

Consider the code below that transposes dictionary keys for values.

```
my_dict = {1:"Red", 2:"Green", 3:"Blue"}
print(my_dict)
print( {value:key for key, value in my_dict.items()} )


>>>
{1: 'Red', 2: 'Green', 3: 'Blue'}
{'Red': 1, 'Green': 2, 'Blue': 3}
```

Conditional Comprehensions

Conditional comprehensions are an excellent way to filter a comprehension. These comprehensions have the code structure:

```
[expression for element in collection if test]
```

In the example below a range of integers from 0 to 9 is generated, but only those with values less than 3 are printed to the screen, owing to the conditional comprehension.

```
print([x for x in range(10) if x < 3])
>>>[0, 1, 2]
```

Conditional comprehensions can also be used in other ways, such as for filtering dictionaries and creating generators, but we shall not discuss this in more detail in this course.

Ternary Expressions

The Introduction to Python course described how to write `if-else` conditional blocs to structure code. For simple expressions, this bloc structure is not necessary however, and can be simplified into a one-line expression. The template for such an expression is:

```
value = true-expression if condition else false-expression
```

Either the `true` expression or the `false` expression will be assigned to `value`, depending on the result of the condition test

Or to give a specific example, see the `if-else` bloc below:

```
x = 2
if x > 1:
    value = "Greater than 1"
else:
    value = "Not greater than 1"
print(value)
>>>
Greater than 1
```

Can be simplified to a one-line conditional:

```
x = 2
value = "Greater than 1" if x > 1 else "Not greater than 1"
print(value)
>>>
Greater than 1
```

Building and using more complex data structures

In the Introduction to Python course we met data structures known as collections. While these are versatile, sometimes they may not meet all your requirements. You may need instead to create a collection that contains other collections. This concept in illustrated below, in which two tuples have been created. These tuples are then added to list, creating a list of tuples.

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)

l1 = [t1, t2]

print(l1)
print(type(l1[1]))

>>>
[(1, 2, 3), (4, 5, 6)]
<class 'tuple'>
```

Hopefully the syntax makes sense. We shan't be describing all the possible permutations for creating such collections of collections, but if you sub-divide the task by creating the inner data structure and then the encapsulating outer data structure (or vice versa) using the syntactic rules already described, you should be able to create any data structure you like. In reality, it is not typical to create data

structures of more than two "levels" of complexity. If you are creating data structures more complex than this, it might be worth re-evaluating whether you have adopted the correct programmatic approach.

The zip() function

The `zip()` function is a useful way to group elements of lists, tuples and other collections to form a list of tuples. The `zip()` function returns a `zip` object, which is an iterator of the combined tuples. By passing this `zip` object to the `list()` function, a list of tuples may be generated. Please note: if the passed collections have **different lengths**, the collection with the **fewest elements** decides the length of the `zip` iterable created:

```
num1 = [1, 2, 3, 4]
num2 = ['One', 'Two', 'Three']

zipped = zip(num1, num2)

print(zipped)
print(list(zipped))

>>>
<zip object at 0x10fe07d20>
[(1, 'One'), (2, 'Two'), (3, 'Three')]
```

A note on scope

We have discussed that assigning names to values and then using these values in subsequent calculations and manipulations. However, just because we have created a name, it does not mean it is accessible to every part of a script. We list some example of how scoping works in Python. These may be particularly interesting for people familiar with other languages (such as Perl) who may get caught out by Python's behaviour.

Variables created in out outer bloc of code will be accessible to the inner bloc of code

You can see below that `h` is in scope inside the `for` loop, since it is successfully added to the value of `i` and then printed to the screen.

```
h = 10
for i in range(2):
    print(h+i)

>>>
10
11
```

Likewise, variables declared outside a function are available within that function:

```
def test_function():
    print(x)
```

```
x = 1
test_function()

>>>
1
```

Loops can modify variables "external" variables, but functions cannot

You see in the example below that, as expected, the toplevel name `x` is accessible within a loop and a function. However, these control structures have different behaviour with regards to modifying `x`.

```
x = 1

for i in range(2):
    x = x + 1
print(x)

>>>
3


def test_function():
    x = x + 1
    print(x)

x = 1
test_function()

>>>
Traceback (most recent call last):
  File "C:\Users\wingetts\Desktop\mypy.py", line 8, in <module>
    test_function()
  File "C:\Users\wingetts\Desktop\mypy.py", line 5, in test_function
    x = x + 1
UnboundLocalError: local variable 'x' referenced before assignment
```

Variables created within a function will not be accessible outside that function

In the two examples below, the code prints the names `x` or `y`, which have been declared within a function. In both cases an error is generated.

```
def add_one(x):
    return(x + 1)
```

```
print(add_one(1))
print(x)

>>>
2
Traceback (most recent call last):
  File "C:\Users\wingetts\Desktop\mypy.py", line 5, in <module>
    print(x)
NameError: name 'x' is not defined


def add_one(x):
    y = x + 1
    return(y)

print(add_one(1))
print(y)

2
Traceback (most recent call last):
  File "C:\Users\wingetts\Desktop\mypy.py", line 6, in <module>
    print(y)
NameError: name 'y' is not defined
```

Variables created in a loop will be accessible outside that loop

In the example below, the names `i`, `j` and `k` are declared within a loop (or a pair of nested loops). Despite this, all these names are accessible to the toplevel coding area.

```
for i in (range(1)):
    for j in (range(10, 11)):
        print(i)
        print(j)
        k = i + j
        print(k)
        print()


print("Outside loop i:" + str(i))
print("Outside loop j:" + str(j))
print("Outside loop k:" + str(k))

>>> %Run mypy.py
```

```
0
10
10

Outside loop i:0
Outside loop j:10
Outside loop k:10
```

Just to make you aware that there is a keyword `global` that can modify Python scoping, but we shall not discuss this further in this course.

An overview of object-oriented programming

The concepts of object-oriented programming

A strength of Python and a feature that makes this language attractive to so many, is that Python is what is known as an **object-oriented programming language (OOP).** (You may occasionally see this written as "orientated" in British English.)

The alternative programming style is procedural, which may be thought of as a set of ordered instructions. Giving someone geographical directions makes a good analogy to procedural instructions: e.g. 1) take the second right, 2) go straight on at the roundabout and 3) turn left at the lights. This style is what most people think of by the term programming and indeed, this is how we have approached programming up until now in this course, since it is a simple and effective way to complete tasks of basic-to-intermediate complexity. As you build more complex programs, however, you may find it becomes ever more difficult to keep track in your own mind as to what is going on. What does a particular function or variable do? How should I arrange my many pages of code? Should I make a value accessible to all parts of my code? These questions you may ask yourself as your codebase increases in size.

OOP is easier for humans to understand, particulary as a program increases with size, because it models our everyday world. That is to say, it categorises its components into objects, which may be thought of as self-contained entities that have their own properties. Different objects may interact with one another and related objects constitute groups know as classes.

In reality, the distinction between an OOP language and a procedural language is somewhat blurred. Perl (previously the most popular bioinformatics language) for example has an OOP component, but it is quite common for even experienced aficionados to hardly ever use this aspect of the language. The statistical programming language R is similar in this regard, but many users will only explicitly deal with R objects when processing the output from external modules. In contrast, Java was designed as OOP from the ground up, and learners will be introduced to these concepts right from the start. Python falls between Perl and Java in that it is quite possible for programmers to write code with only a passing familiarity with objects, such as when executing methods on particular objects. However, with a little bit more experience it is quite possible to build complex object-orientated software in a style more typical to Java.

In this advanced course we shall introduce the key concepts of OOP, yet steer away from an in-depth explanation of the subject. For more details on OOP, please consult the Babraham Bioinformatics course dedicated to this area of the Python language.

Objects in Python

Simply put, an **object** is a data structure that has values and associated methods. In Python, almost everything is an object: a string for example stores values, and may be queried or manipulated by a number of methods. In addition to objects native to the Python language, a programmer may create their own novel objects. More specifically, a programmer would actually define a **class** (a group of related objects). For example, a particular dog is but an **instance** of the dog class.

[The following sentences may sound strange at first, but please bear with me.] If we wanted to create a dog in our program, we would define the dog class, and then make a *specific* dog from that class. Each dog would constitute a separate Python object, in a sense modelling the real world. (Technically speaking, in Python even the abstract concept of a class is an object in its own right.)

To understand further the basics of building a class, please look at the code below. You don't need to understand everything in the code, but just get an idea of how the code is structured. What is more important is that you understand how to interact with an object in the main body of your code (i.e. create an instance of an object and use object's methods).

```
class Dog:
    def get_mood(self):
        return self.mood

    def set_mood(self, data):
        self.mood  = data

    def animate(self):
        if self.mood == 'Happy':
            return('Wag Tail')
        elif self.mood == 'Angry':
            return('Bite')
        else:
            return('Bark')

snoopy = Dog()
snoopy.set_mood("Happy")
print(snoopy.get_mood())
snoopy.set_mood("Sad")
print(snoopy.mood)
snoopy.set_mood("Angry")
print(snoopy.get_mood())
print(snoopy.animate())

>>>
Happy
Sad
```

```
Angry
Bite
```

Firstly, we would define our dog class using the keyword `class`. To make an instance (`snoopy`) of the dog class, the class is called (as one would call a function):

```
snoopy = Dog()
```

Instances of a class may have **methods** (such as already seen with built-in objects) and store information in what is known as **fields**. Collectively, methods and fields are known as **attributes**. Both of these may be accessed using the now familiar dot notation.

In the example, you see that the field `mood` is set using the `set_mood` method. We firstly set Snoopy's mood to "Happy". The `mood` field can either be accessed from the main script via the `get_mood()` method, or directly on the instance snoopy, using dot notation (`snoopy.mood`).

If you look at the `Dog` class some more, you will notice that each method is defined internally within the class as a function. The `animate` method for example, corresponds the `animate` function. The code within this function should be quite familiar, for it comprises a series of conditionals taking a dog's mood as input.

Inheritance

The concept of **inheritance** is central to object orientated programming and allows programmers to write code much more efficiently. The rationale owes much to the phenomenon of the same name observed in biology, in which organisms with a certain set of traits produce offspring with largely the same characteristics. In OOP, once we have defined a class, we can easily define a **subclass** that automatically "inherits" the code of its parent class (now referred to as the **superclass**). We can then change the properties of the subclass, so while it resembles the superclass in many ways, it also has its own distinct functionality.

This ability of OOP is advantageous as it allows coders to produce objects (remember, all classes are objects) with a wide range of functions with a much-reduced code base. It also prevents duplication of code, which is good since if we subsequently need to make changes, we should only have to make the modification in one place and not in many different locations. The process of inheritances may take place over many generations i.e. it is possible to make a subclass and then make a subclass of that.

OOP Summary

So, to summarise:

1) Python is an object orientated programming language, which means almost everything in Python is an object.
2) Objects are data structures with values and methods.
3) Programmers can make custom Python objects.
4) Related objects constitute classes.
5) A specific version of a class in an instance.
6) The data stored in an instance (fields) may be accessed or modified using methods (or even directly using the dot operator).
7) Sub-classes may be created from a class via a process termed inheritance.

Generators – a better way to generate data

We previously alluded briefly to the Python object known as a generator, which produces a stream of data. In many cases this is preferable to an alternative strategy of keeping a very large list in memory, and then selecting a value from the list as required. The syntax for making a generator is almost identical to that of declaring a function, except that code defining a generator ends with the keyword `yield` instead of `return`. When called, a generator object is returned. The keyword `next` instigates each iteration of the generator object.

In the following example we create a generator that produces even numbers. Each time the `next` command is run in the loop, the generator yields the next even number in the sequence. Importantly, notice that once we leave the loop, but call the generator once more, we produce the next even number in the sequence (i.e. the generator keeps track of "where it has got to").

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2

even_number_generator = all_even()

for i in range(5):
    even_number = next(even_number_generator)
    print(even_number)

print("And again...")
print(next(even_number_generator))

>>>
0
2
4
6
8
And again...
10
```

Generators don't have to produce an infinite number of values (they do in this example, because we have constructed an infinite `while` loop). We could specify some a limit to the number of values that can be returned (e.g. only produce numbers while `n` is less than a specified value). Once this is reached, further iteration will not be possible and a `StopIteration` error will occur.

It is also possible to pass arguments to generators in the same fashion as when passing arguments to functions:

```
def my_generator(argument1, argument2, …):
```

So to recap, generators constitute a clear and concise way to make data. They are memory efficient as compared to regular functions that need to create an entire sequence in memory before returning the result. Owing to these reduced memory overheads, generators are generally more efficient in such instances.

Exception handling

No doubt by now when trying to run code you have written, the Python interpreter will have responded by outputting to the screen a **traceback** message of some kind, meaning that there is some kind of error in your code. It is worth pointing out that taking the time to read the traceback message is certainly worthwhile, as the output gives an explanation why the code failed. Although it not always obvious what the output means, with time and experience the traceback message should help you identify most bugs (coding errors) in a relatively short amount of time. Despite any initial misgivings you may have, you will soon find out that the traceback message is the Python coders' friend.

Even if you write perfect code, however, there are events which will be entirely beyond a programmer's control. Suppose you write a script that takes input from a file, but unfortunately, for whatever reason, that file was deleted from the computer running the Python script and so when the program tries to open the file, an error will result. However, a good programmer will plan for those eventualities and although you cannot stop your script failing, when it does fail, you can ensure it fails as gracefully as possible.

Errors can be categorised into different types: trying to open a file that is not on your system you would generate an `IOError`. In contrast, a `NameError` would result if trying to use a name that had not yet been declared within your script.

Fortunately, Python has a feature available to a programmer to deal with such unexpected errors. This capability is known as **exception handling**, and enables a programmer to provide instructions to the script to be on the "look out" for certain types of error, and provides a "contingency plan" of what to do if such an error is encountered. So, an **exception is an error that is handled by the Python code**, and will be dealt with in a pre-specified manner. **Unhandled exceptions** are errors and will result in a **traceback** message.

The basic way to deal with errors is to use a `try` statement followed by an `except` clause. Listed below is a standard template for handling exceptions:

```
try:
    statements we try to run
except ErrorClass:
    what to do if an error of this class is encountered
```

Let's illustrate this with an example:

```
my_list = ['A', 'B', 'C']
print(my_list[3])

>>>
Traceback (most recent call last):
```

```
  File "C:\Users\wingetts\Desktop\thonny.py", line 4, in <module>
    print(my_list[3])
IndexError: list index out of range
```

We have generated a “list index out of range error” by trying to get the value at index 3 from `my_list`. Although `my_list` has 3 elements, the numbering of indices starts at 0, so there is no value at position 3 in our list. We could write an exception handler for this:

```
try:
    my_list[3]
except IndexError:
    print("That index is outside the list!")


>>>
That index is outside the list!
```

Now, thanks to the exception handler, the code does not terminate with a traceback, but rather prints to the screen a user-friendly description of what went wrong. Furthermore, if this code were part of a larger program, the program should carry on running while conversely, without the exception handler, the program would terminate abruptly.

Many error handlers are more complex than the example above, catching many different types of errors, and potentially handling each type of error differently. Look at the following template:

```
try:
    statements we try to run
except ErrorClass1:
    what to do if an error of class 1 is encountered
except ErrorClass2:
    what to do if an error of class 2 is encountered
except (ErrorClass3, ErrorClass4):
    what to do if an error of class 3/class 4 is encountered
except ErrorClass5 as err:
    what to do if an error of class 5 is encountered
except:
        statement to execute if error occurs
finally:
        statement carried out if exception is/is not made
```

The above template shows that the error handling reporting process can be tailored to the type of error generated, for example `ErrorClass1` and `ErrorClass2` will result in different responses. `ErrorClass3` and `ErrorClass4` will generate the same response. The response to `ErrorClass5` generates an error object called `err` using the `as` keyword. This object may then be processed by the script as required. The statement following the last `except`, in which the error type is not defined, will be performed if an error takes place, but has not been specified previously in the error handling block

of code. The `finally` statement will be carried our regardless, irrespective of the type of error, or indeed if any error, has occurred. To illustrate a more complex error handling process, look at the code below:

```
my_dict = {"Aardvark":1, "Baboon":2, "Cougar":3}
try:
    value = my_dict["Dandelion"]
except TypeError:
    print("This index is unhashable")
except KeyError:
    print("This key is not in the dictionary!")
except:
    print("Some other error occurred!")
finally:
    print("Let's carry on")

>>>
This key is not in the dictionary!
Let's carry on
```

The code creates a dictionary containing three values which are animals, but then tries to retrieve a value (i.e. “Dandelion”) not in the dictionary. This returns a `KeyError` leading to message “This key is not in the dictionary!” being printed to the screen. The `finally` clause is then executed after the rest of the code.

(While Python has a wide range of built-in errors, when developing more complex code there may be times when you need to define custom errors. It is possible to write code to create an **exception object**, which may be returned should an error occur. We shall not cover this in this course, but so you are aware: exceptions in Python are instances of the built-in class `Errors`. To create your own error, you would need to import that class and then define your custom error as a subclass.)

Chapter 2 – modules and packages

Introducing modules and packages

By using the Python techniques already described in this course, and by making use of the language's built-in functions and methods, it is possible to perform a wide variety of calculations and manipulations with a small amount of code. This capability is extended substantially by making use of **modules** and **packages**.

(Note: strictly speaking a module is a single Python file that can be imported, while packages comprise multiple Python files and can even include libraries written in different languages. It is not uncommon for these terms to be used interchangeably and simply refer to the anything that extends the functionality of a Python script by using the `import` command.)

Modules make available to the user a much greater range of data types, functions, and methods. Under the hood, modules are actually files placed in a library directory during Python installation. Modules can also be obtained from external sources and added to Python's library. There are also a wide variety of bioinformatics modules available that can be installed to assist with your data analysis. To import a module, use the import command:

```
>>> import os
```

This will import the `os` module, which allows Python to communicate with your operating system. Now the `os` module is installed it is possible to run functions within this module:

```
>>> os.getcwd()
'/Users/wingetts'
```

The `getcwd()` function returns the user's current working directory. To call a function within a module, enter the module name (e.g. `os`), followed by a dot, then the required function's name (e.g. `getcwd`) and then a pair of round brackets as above.

Sometimes you may want to just import specific names from a module, rather than everything found within that module. The following example shows how to import only `getcwd` from the `os` module. Use this syntax for other importing specific names from other modules.

```
from os import getcwd
```

Importing only what you need often makes more sense as it can be overkill to import a whole module simply for one function, or similar. Importing in this way means you can refer to `getcwd` directly, rather than by using `os.getcwd` (although either syntax is acceptable). Anyway, don't worry about this distinction, the take home message is that there are many modules available that can be imported by a script to instantly extend the functionality of that script. This chapter discusses some of these modules and once you have become familiar with these, you should be able to use a wide variety of additional modules. Some of these modules have bioinformatics functionality, while others are linked solely computational in nature, extending the capabilities of a Python script, or how it interacts with your computer.

In Thonny, it is possible to import modules using the package manager which can be accessed via the main menu bar: Tools -> Manage Packages.

The datetime module

Some programs may need to retrieve the current time and date information, or they may need to manipulate times and dates in some way. The **`datetime`** module was written specifically for such tasks. Suppose then that you wanted to use this `datetime` module, how would you go about using it? Well, in the first place you should check the documentation for the module, which can be found at: https://docs.python.org/3/library/datetime.html

The documentation is quite technical and there will undoubtedly be parts that you don't understand, but with growing familiarity and practice you will be able to understand much of what is being described and extract from the document all that is required to write your own code. The text below is taken from the documentation (for Python 3.2). The text lists the classes contained within the `datetime` module.

Available Types
class datetime.date
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class datetime.time
*An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. (There is no notion of "leap seconds" here.) Attributes: hour, minute, second, microsecond, and tzinfo.*

class datetime.datetime
A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class datetime.timedelta
A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class datetime.tzinfo
An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.timezone
A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

The module is versatile and enables the user to perform a wide variety of calculations. Here are some examples of how to use the module to achieve some commonplace date/time-related tasks. In this first example we have imported the `datetime` module, we then instantiate a `datetime` object using the current time as time stored by the object. This value is then printed to the screen as a single-line timestamp, using the `print` function.

```
import datetime


my_datetime_object = datetime.datetime.now()
```

```
print(my_datetime_object)

>>>
2019-12-16 17:40:35.218147
```

It is worth pointing out the all the attributes of a module can be achieved using the `dir` function. You should see that the resulting list of attributes corresponds to that listed in the documentation

```
import datetime

print(dir(datetime))

>>>
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

It is also possible to create a date object that corresponds to a user specified date:

```
import datetime
my_date = datetime.date(1966, 7, 30)
print(my_date)

>>>
1966-07-30
```

In addition, the module can be used to add or subtract dates to determine the amounts of time accumulated in different situations. In the example below, we import the `datetime` and `date` classes. We then create to date objects, set to dates of our choosing. Then, by simply using the minus operator, we are able to deduce the length of time between them.

```
from datetime import datetime, date

my_start = date(year = 2016, month = 7, day = 12)
my_end = date(year = 2019, month = 7, day = 24)
my_term = my_end - my_start
print(my_term)

>>>
1107 days, 0:00:00
```

As mentioned before， when you want to perform a task with an additional module, there is often some degree of research involved. Firstly, you may need to find out if a module that meets your programming requirements actually exists. Typically, programmers will refer to modules they have used previously

to see if their required functionality is available. If that proves not to be the case, it is common then to search the Python documentation or using Google (or another favourite search engine) to find a module that appears suitable. Next will be the phase of reading the module's documentation to see how that particular module works, and then finally trying the module out and incorporating it into the codebase.

The exercises for this chapter were designed to help you go through this process yourself. The rest of the chapter briefly discusses other useful modules and how they are typically used. We shall not go into detail with regards to their implementation, since this will be handled as part of the exercises.

The math module

As the name implies, this module provides many trigonometric and hyperbolic functions, powers and logarithms, angular conversions and well as mathematical constants such as `pi` and `e`. Further information on this module can be found at: https://docs.python.org/3/library/math.html. This module should be your starting point if you wish to perform some kind of mathematical operation beyond that available to you in standard Python. With what you know already, much of the syntax of the module should be relatively straight forward to pick up, such as the following code used to calculate the base-10 logarithm of 1000.

```
import math

print(math.log10(1000))
```

```
>>> %Run test.py
3.0
```

The sys module

The **`sys` module** contains information pertaining to the Python implementation being used. Below is a list of commonly used values and examples of the output you would expect to see if you print them to the screen.

sys.argv

Description: this name is a list of strings containing the elements of the command line used to run the script. As we might expect, the first element of `sys.argv` (`sys.argv[0]`) reports the name of the script being run.

Printing to the screen:

```
['thonny.py']
```

sys.modules

Description: This is a dictionary in which the keys name the currently loaded modules.

Printing to the screen:

```
thonny.py ('_abc', '_ast', '_bisect', '_blake2', '_codecs', '_codecs_cn',
'_codecs_hk', '_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_codecs_tw',
'_collections', '_csv',…
```

The first part, thonny.py, is the name of the script and then the modules are listed between the brackets.

sys.path

Description: A list of the directories into which Python looks for modules following the `import` command in a script.

Printing to the screen:

```
['C:\\Users\\wingetts\\Desktop',
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\python37.zip',
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\DLLs',
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\lib',
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny',
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\lib\\site-
packages']
```

sys.exit()

This is a function that will cause a Python program to exit. It is the standard practice to pass the argument 0 if the program exits without error. Numerical values other than 0 signify some kind or error. The `sys.exit()` function is preferable in scripts to the `exit()` or `quit()` commands, which are only intended for use when using the interactive Python interpreter.

The time module

Although the time module contains many time-related functions, you are most likely to encounter the function: `time.sleep()`. This instructs the Python script to wait for the specified number of seconds passed as an argument to the function. Why would you want to do this? Well, sometimes slowing down a program is a useful way to check output written to the screen at a pace that is readable by a human. In addition, sometimes we may add a delay to a script to ensure some other process has finished before we run the next part of the script.

The argparse module

Maybe you have seen already when running scripts on the command line that they may take as input a series of options, each one prefixed by one or two hyphens. Python scripts may also be passed options this way, and this is made possible by using the **`argparse`** module. The module enables the user to specify what flags may be used be a script, what (if any) argument these flags take, and the type of arguments associated with each flag. For example, the flag `--length` may only take integers while the flag `--mode` may take a string. What is more, the `argparse` module will automatically generate instructions regarding the script and its options, which can be viewed by specifying `--help` (or `-h`) when running the script.

```
python3 myPythonScript.py --length 500 --mode fast file1.txt file2.txt
file3.txt
```

There are many ways in which to setup and use `argparse` and we recommend looking at the official Python documentation to learn how to implement this module

The subprocess module

The command line is a versatile and convenient environment in which to manipulate your system and run scripts and software. Since pipeline development (the joining separate processes into a single workflow) is a common use of Python for bioinformaticians, it would be useful to incorporate some of

the functionality of the command line into a Python script. The **subprocess** module makes this possible by allowing a shell command to be executed directly from a Python script.

For example, the following Python script could be run directly from the command line, and will print "Hello World" as output.

```
import subprocess

print(subprocess.getoutput('echo "Hello World!"'))

---

user$ python3 subprocess_example.py
Hello World!
```

The os module

The `os` module is an operating system interface and is most commonly used in Python scripts for interacting with files and directories in the filesystem. The selected example below illustrates how the `os` module may be used to interact with your filesystem.

`os.chdir(path)`
Sets path to the working directory

`os.getcwd()`
Lists the current working directory

`os.mkdir(path)`
Creates a directory at specified location

`os.mkdirs(path)`
Creates all the directories in the path specified

`os.rmdir(path)`
Removes the specified directory

`os.removedirs(path)`
Removes all the directories in the path specified

`os.remove(path)`
Delete the specified file

`os.rename(sourcepath, destpath)`
Rename the file at `sourcepath` to `destpath`

`os.path.dirname(path)` Return the directory name of pathname path. So, if `path = '/home/User/Desktop/myfile.py'`, then `'/home/User/Desktop'` would be returned.

`os.path.basename(path)` Returns the basename of the path. So, if `path = '/home/User/Desktop/myfile.py'`, then `'myfile.py'` would be returned.

The tempfile module

In the future you may write a script that during processing writes out a temporary file to a specified directory. This may sound straightforward, but what happens if you are running multiple instances of the same script? There is a danger that one instance of the script could overwrite a temporary file created by another instance of the script. The solution may seem simple, such as numbering the temporary output files. However, in reality, these solutions that may seem logical can fail and guaranteeing that output files from one process are not overwritten by another process is not a trivial process. Fortunately, the **tempfile** module handles this all for you.

The glob module

Suppose there is a file or a list of files on your computer that needs processing in some way. Unfortunately, you don't know the names of the files in advance (you may only know, for example, that the files are found in a certain folder). You will therefore need some way for your script to search the filesystem, return the relevant filenames and process these. The **`glob`** module allows you to do this.

To use `glob`, simply specify the pattern you wish to match:

```
glob.glob(pattern)
```

The operation of the pattern matching is similar to using the command line in that it also allows **wildcards**:

`*`	match 0 or more characters
`?`	match a single character
`[agct]`	match multiple characters
`[0-9]`	match a number range
`[a-z], [A-Z], [a-Z]`	match an alphabet range

In the simple example below, the script will identify all files is the current working directory, and then all text files in that folder. The `glob.glob` command generates a list, which is then printed to the screen.

```
import glob

all_files = glob.glob('*')
text_files = glob.glob('*.txt')
print(all_files)
print(text_files)

>>>
[''make a better figure.pptx', 'one_hundred_lines.txt',
'subprocess_example.py']
['one_hundred_lines.txt']
```

The textwrap module

Writing code to format text in a consistent, neat and logical manner is surprisingly difficult, but once again Python modules come to the rescue. The **`textwrap`** module does just that, providing a list of functions to format text in a variety of ways.

Perhaps the most commonly used components of the `textwrap` module are the **`textwrap.wrap`** and the **`textwrap.fill`** functions that serve as convenient ways to handle, format and print out long strings, such as encountered in FASTA files. In the example below you will see how a lengthy string may be converted into a list by `textwrap.wrap`, which may be subsequently printed out using a `for` loop. In contrast, the input may be formatted into a new string using `textwrap.fill`. Both these functions take optional integer arguments specifying the maximum character length of each line.

```
import textwrap

quote = """It was the best of times, it was the worst of times, it was the
age of wisdom, it was the age of foolishness, it was the epoch of belief,
it was the epoch of incredulity, it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we were all
going direct to Heaven, we were all going direct the other way - in short,
the period was so far like the present period, that some of its noisiest
authorities insisted on its being received, for good or for evil, in the
superlative degree of comparison only."""

formatted_quote = textwrap.wrap(quote, 50)
for line in formatted_quote:
    print(line)

print("\n")

print(textwrap.fill(quote , 70))


>>> %Run subprocess_example.py
It was the best of times, it was the worst of
times, it was the age of wisdom, it was the age of
foolishness, it was the epoch of belief, it was
the epoch of incredulity, it was the season of
Light, it was the season of Darkness, it was the
spring of hope, it was the winter of despair, we
had everything before us, we had nothing before
us, we were all going direct to Heaven, we were
all going direct the other way - in short, the
```

```
period was so far like the present period, that
some of its noisiest authorities insisted on its
being received, for good or for evil, in the
superlative degree of comparison only.

It was the best of times, it was the
worst of times, it was the age of
wisdom, it was the age of foolishness,
it was the epoch of belief, it was the
epoch of incredulity, it was the season
of Light, it was the season of Darkness,
it was the spring of hope, it was the
winter of despair, we had everything
before us, we had nothing before us, we
were all going direct to Heaven, we were
all going direct the other way - in
short, the period was so far like the
present period, that some of its
noisiest authorities insisted on its
being received, for good or for evil, in
the superlative degree of comparison
only.
```

The string module

The **`string` module** (not to be confused with the `str` data type) contains a range of useful values and functions. For example, it contains all the upper- and lowercase letters in the names **`string.ascii_uppercase`** and **`string.ascii_lowercase`** respectively. There are also names representing digits, punctuation and a range of other string values.

The csv module

If you have ever worked with data file storing text values, the chances are you will have encountered the **comma-separated values (CSV)**. This format stores a table or matrix (with layout similar to that of *MS Excel*), only it uses commas to separate columns. (As may be expected, rows are separated by newlines.) A very similar format uses tabs instead of commas to separate columns; this is known as **tab-separated values** (or **tab-delimited**) format. The **`csv` module** enables a Python script to read from or write to these types of file. Again, the official Python documentation should be consulted for guidance on how to used this module.

The zlib and gzip modules

Modern life science data files are often are very large. To assist with their storage, they are often compressed as either zip or gzip files. The `zlib` and `gzip` modules are available for reading from or writing to these file types, respectively. The syntax for their usage is generally similar to that for opening a regular file. One important difference is that data returned from the file will be a sequence of `bytes` (note the `b` before the quotation mark when printing to the screen in the example below.) We have not discussed this `bytes` datatype before, but simply put it is a machine-readable format that can be directly

stored on the disk (in contrast to strings, which are in human-readable but cannot be saved directly to disk). The `bytes` datatype can be decoded to a string with the decode method `decode('utf-8')`. UTF-8 is the type of Unicode format to which the `bytes` should be decoded.

```
import gzip

with gzip.open('test_file.txt.gz', 'rb') as f:
    file_content = f.read()

print(type(file_content))
print(file_content)
print()
file_content_string = file_content.decode('utf-8')
print(type(file_content_string))
print(file_content_string)
```

```
<class 'bytes'>
b'Hello.\nI am a compressed file.\n'

<class 'str'>
Hello.
I am a compressed file.
```

When writing to binary files, create a file object with the parameter `'wb'` passed to `gzip.open` (instead of `'rb'`- used for reading files).

Although this section is now at an end, this does not mean that we covered all the in-built Python modules. We have tried to identify the modules that will be most useful to you, but as you write more sophisticated Python, you should bear in mind that the code you need to complete a particular task may only be a simple import away. With that in mind, it is a good idea to spend the odd occasion reading more about Python modules in the official and documentation and elsewhere to improve your coding proficiency.

Installing Modules and Packages

Up until now the modules discussed are distributed with Python and so can be simply be made accessible via the `import` command. There are many modules, however, particularly in the fields of bioinformatics or some other specialist area of biology, that require downloading and installation. If all goes well, this process should be quite simple thanks to the **pip** installer program, which is included by default with Python binary installers since version 3.4.

Installation

The following command shows how to use pip to install the latest version of a **package** (a collection of modules) and its dependencies:

```
python3 -m pip install SomePackage
```

If you have Python2 installed, you may need to type “python3” instead of “python” to clarify the version of python that should be used; we used python3 here for clarity. The flag `-m` instructs the python to run the module `install`. After running this command, and if everything proceeds successfully, you should see a message similar to: “Successfully installed SomePackage-1.2.3”. (It is possible to install packages using the pip3 command directly, but we shall use the methods described above.)

It is possible to specify an exact or minimum version directly on the command line. To install a specific version (say 1.1.2) try:

```
python3 -m pip install SomePackage==1.1.2
```

To specify a minimum version (say, 1.0.2), try:

```
python3 -m pip install "SomePackage>=1.0.2"
```

(Don’t forget the double quotes as the character “>” may cause unintended behaviour when run on the command line.)

Should the desired module be already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python3 -m pip install --upgrade SomePackage
```

It is worth bearing in mind that up-to-date copies of the **`setuptools`** and **`wheel`** projects are useful to ensure you can also install from source archives. To update these, run the command below:

```
python3 -m pip install --upgrade pip setuptools wheel
```

You can read up some more on `setuptools` and `wheel` files if you wish, but all the typical coder needs to bear in mind is that these facilitate the installation of packages.

Installation locations

So, where are Python packages installed? Well, to find this out, type the command:

```
python3 -m site
```

This should return information, similar to that listed below:

```
sys.path = [
    '/Users/wingetts',
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python38.zip',
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8',
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/lib-
dynload',
    '/Users/wingetts/Library/Python/3.8/lib/python/site-packages',
```

```
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-
packages',
]
USER_BASE: '/Users/wingetts/Library/Python/3.8' (exists)
USER_SITE: '/Users/wingetts/Library/Python/3.8/lib/python/site-packages'
(exists)
ENABLE_USER_SITE: True
```

The Python packages will be stored in the `sys.path` section, in the path ending `site-packages`. If you take a look in that folder, you should see your installed packages.

It is quite likely, however, that on the system you are using, you will not have administration rights to install the package you desire i.e. you may not write to the `site-packages` folder listed above. Not to worry, you should be able to install packages that are isolated for you by adding the `--user` flag:

```
python3 -m pip install SomePackage --user
```

This package will now be installed in the `site-packages` folder, to which you will have access, listed in the `USER_SITE:` section.

Using `pip` is the most common way to install modules and should work in most instances. However, not every software developer will make their tools available via pip. In such circumstances, the desired module should give instructions regarding how it should be installed.

Virtual Environments

Although not described in this course in detail, you should be aware of **Python virtual environments** that allow Python packages to be installed in an isolated location for a particular application, rather than being installed globally. This means that you can install Python packages expressly for one (or several) particular applications without changing the global setup of your system. You then simply start the virtual environment when trying the application of interest, and then exit when done.

Biopython

Biopython is a set of freely available Python tools for biological computation and as such provides a useful resource for the life sciences community. This may be one of the packages that are not distributed with Python that you will need to install to help you complete some computational task. Of course, with your new-found skills in Python you may be able to write the necessary application yourself, but there is no point re-inventing the wheel and if what you require is already available using Biopython, then this should be a good starting point for your analysis. The Biopython homepage is found at: https://biopython.org/

As you might expect, to install the package, enter on the command line one of the following commands (in accordance with your administration privileges):

```
python3 -m pip install biopython
python3 -m pip install biopython --user
```

Below is an example given in the Biopython documentation, which shows how the `SeqIO` standard Sequence Input/Output interface for Biopython may be used to parse a Genbank format file into a FASTA file. In a relatively small amount of code a useful task can be accomplished, highlighting how Python and its large, active community of developers can be harnessed so you can achieve tasks much more efficiently than if you were starting writing code from scratch.

```
from Bio import SeqIO

with open("cor6_6.gb", "r") as input_handle:
    with open("cor6_6.fasta", "w") as output_handle:
        sequences = SeqIO.parse(input_handle, "genbank")
        count = SeqIO.write(sequences, output_handle, "fasta")

print("Converted %i records" % count)
```

Chapter 3 – regular expressions

Sequence data – whether that is derived from DNA, RNA or protein samples – constitutes an ever more important and burgeoning area of research. Identifying, extracting or modifying specific sequences from the vast pools of data that is commonly generated in modern experiments is no trivial task. Fortunately, the world of computer science has already developed a toolkit with the pattern matching capabilities suited to this formidable challenge. These tools, which constitute a language in their own right, are known as **regular expressions** (or more familiarly known as **regexes** or **RE**s). There is a substantial amount of new and dense material to cover in this chapter, so this section may take some effort and practice to become confident at pattern matching. But do take heart, even though regexes may first appear to be impenetrable lines of hieroglyphics, once you can use them, they will form a powerful and succinct way to scrutinise sequences. Moreover, the same ideas and notation are used in different programming languages (e.g. R, Perl and Java), allowing you to also build skills outside of Python.

Introducing the re module

The Python **`re` module** provides an interface with the regular expression engine, allowing you to compile regexes and then perform matching operations.

As you may expect, to use the re module, simply enter `import re` towards the top of your Python script.

Simple String Matching with the re Module

So now we have introduced the `re` module, let's try some simple pattern matching of strings. Take a look at the code below in which we intend to identify the HindIII restriction enzyme cut site (AAGCTT) in three different sequences.

```
import re

pattern = 'AAGCTT'
p = re.compile(pattern)

seq1 = 'AAGCTTNNAAGCTT'
seq2 = 'GGGGGG'
seq3 = 'NNAAGCTT'

print(p.match(seq1))
print(p.match(seq2))
print(p.match(seq3))
print(p.search(seq3))

>>>
<re.Match object; span=(0, 6), match='AAGCTT'>
None
None
<re.Match object; span=(2, 8), match='AAGCTT'>
```

We firstly import the `re` module. After that, we then compile the regular expression (the pattern we wish to identify) into a **complied pattern** object (named `p`). We now run the match method on this object, taking `seq1`, `seq2` and then `seq3` as arguments. The first time we do this (with `seq1`), we achieve a match and a **match object** is returned. We can tell this has happened by printing out this newly generated object, for we can now see displayed the match string and the position of the match. In contrast, no match was achieved for `seq2` and `seq3`, resulting in `None` being retuned. It is quite clear that `seq2` does not contain the HindIII site, but `seq3` does – so why wasn't a match object returned? Well, the method `match` only matches at the *start* of sequence. In contrast, the `search` method allows matches *anywhere* in the sequence, and so the match was successful.

The table below lists the compiled pattern object methods.

Method	Purpose
`match()`	Determine if the regex matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this regex matches.
`findall()`	Find all substrings where the regex matches, and returns them as a list.
`finditer()`	Find all substrings where the regex matches, and returns them as an **iterator of match objects** (i.e. the match object may be retrieved one at a time using a `for` loop)

Querying the match object

The match object returned after a successful match has methods which may be executed to retrieve useful information, as listed in the following table.

Method	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match

The code printed below illustrates this ability to query the match object with these methods. The code is a modification of the previous example, and here we use search to identify a HindIII site within our DNA sequence. We then print matched sequences, the start position of the match, the end position of the match by retrieving this information from the match object.

```
import re

pattern = 'AAGCTT'
p = re.compile(pattern)

seq = 'NNAAGCTT'

m = p.search(seq)
print(m.group())
print(m.start())
```

```
print(m.end())

>>>
AAGCTT
2
8
```

You may be wondering at this point as to the usefulness of being able to print the matched sequence; after all we already know the identity of our sequence, since we passed this to our regular expression object to achieve the match. Well, that is true for this simple example, but often we will not know the exact sequence that will be matched in advance since we may be trying to match a range of different sequences. In the next sections we shall describe how to construct these more complex regex look-up terms.

Metacharacters

There are special characters, termed **metacharacters**, in regular expressions that represent other types of characters. The list of metacharacters is:

```
. ^ $ * + ? { } [ ] \ | ( )
```

We shall now provide an overview of how these are used.

Character classes

The metacharacters `[` and `]` are used to define classes of characters. For example [abc] means match the letters a, b or c. The character class `[a-c]` will achieve the goal. Similarly, you may match against all lowercase characters with the class `[a-z]` or against any digit with `[0-9]`. Metacharacters generally lose their special properties inside the square brackets and instead become literal representations of themselves.

The caret character ^ is used to complement the set. So, for example, `[^7]`, will match any character *except* 7.

On a related note, the pipe character (`|`) essentially means 'or' within a regex. The pattern `a|b` therefore will constitute a match on either a or b.

Start and ends

When found outside of the character class, the caret (**^**) denotes the start of the string. Similarly, the dollar symbol (`$`) denotes the end of a string. So, for example, the regex `^ABC$` would mean a matching pattern should contain ABC and have nothing either side (i.e. the start character is A, and the end character is B). Or for example, the regex `^pi` would match `pi`, `pike`, and `pill`, but not `spill`. This is because a successful match requires the string to *start* with a `p`.

Backslash

The backslash is commonly used in regular expressions and serves multiple purposes. Firstly, the backslash is often followed by a letter character, and together those two characters represent a whole range of different characters. As seen before, the character pair `\d` represents numerical (base-10) digits. The table below describes the backslash letter pairs.

Backslash Letter	Meaning
\d	Matches any decimal digit; this is equivalent to the class [0-9]
\D	Matches any non-digit character; this is equivalent to the class [^0-9]
\s	Matches any whitespace character; this is equivalent to the class [\t\n\r\f\v]
\S	Matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v]
\w	Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_]
\W	Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9_]

The above pattern matches may be used inside character classes to increase their power.

Another very useful metacharacter is the **dot (.),** which matches any character, except newline characters.

Escaping Using Backslashes

You may have noticed a problem with the regular expression we have introduced so far. One such problem involves using the dot (.) as a metacharacter. While this is undoubtedly useful, what happens if we want to explicitly match a full stop (or period)? We can't use this character itself, since it will match everything. The solution is a technique known as escaping. Preceding a metacharacter with backslash will cause it to be interpreted literally, rather than as a metacharacter. For example:

```
pattern1 = '.'
pattern2 = '\.'
```

The regular expression `pattern1` will match everything except new line character, while `pattern2` matches full stops. This "**escaping**" technique may be used for other metacharacters, even the backslash itself, as shown that `pattern3` matches the backslash character itself.

```
pattern3 = '\\'
```

Raw String Notation

An alternative to escaping characters (which may become very difficult to read) is to use **raw string notation**. This is quite easy to implement and all that one needs to do is place the character `r` before the string for it be interpreted literally.

For example, to read the text `\\matchme` literally, use the notation: `r"\\matchme"`.

Repetition

Regular expressions allow the user to denote that a phrase needs to be repeated a specified number of times for a match to occur. Specifically, the metacharacters **asterisk (*), plus (+)** and **question mark (?)** achieve this in closely related, but slightly different ways (see the table below).

Metacharacter	Action
*	the preceding character should be occur **zero or more times**
+	the preceding character should be occur **one or more times**
?	the preceding character should be occur **zero or one times**

The small sample code below should illustrate this functionality. The code creates a regular expression to match the pattern `CAR*T`. As should be expected, matching against the letters `CART` will generate a successful match. Essentially the required pattern is `CART`, although the letter `R` may be "present" zero or more times. In the second pattern match we test whether the word `CAT` matches the specified criteria. It does, since the asterisk means the `R` may be omitted. Removing the letter `R` from `CART` makes `CAT`, and so we create a match object.

```
import re

pattern = 'CAR*T'
p = re.compile(pattern)

letters1 = 'CART'
letters2 = 'CAT'

m1 = p.search(letters1)
m2 = p.search(letters2)

print(m1)
print(m2)

>>> %Run regext.py
<re.Match object; span=(0, 4), match='CART'>
<re.Match object; span=(0, 3), match='CAT'>
```

Another method to denote repetition involves placing an integer between curly brackets: `{n}`. This denotes the exact number of times the preceding character should be displayed. Placing two integers between these brackets is another way to specify repletion: `{m,n}`. This repeat qualifier means there must be at least `m` repetitions, and at most `n` of the preceding character. For example, `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`. It won't match `ab`, which has no slashes, nor `a////b`, which has four. In fact, you don't need both the *m* and *n*, since by default omitting *m* is interpreted as a lower limit of 0 `{ , n}`, while omitting *n* results in an upper bound of infinity `{m, }`.

So, we have at our disposal different techniques to denote the repetition of individual characters. But these techniques are even more powerful, since they may **used to denote repetition of character classes (rather than simply individual characters).**

```
import re

pattern = '[aeiou]+'
p = re.compile(pattern)

word = 'euouae'

match = p.search(word)

print(match)

>>>
<re.Match object; span=(0, 6), match='euouae'>
```

Greedy vs non-greedy matching

In addition to the techniques of specifying repetition, there are a set of related techniques that work in the same way, except these perform non-greedy matching as opposed to greedy matching. The non-greedy qualifiers are the same as those for greedy matching (described previously), except that they are followed by a question mark:

```
*?
+?
??
{m,n}?
```

Greedy matching matches as many characters and possible, while nongreedy matching matches the fewest allowed characters. Please review the code below to see how adding the qualifier `?` to the pattern `Pneu.*s` modifies its matching behaviour.

```
import re

pattern_greedy = 'Pneu.*s'
pattern_nongreedy = 'Pneu.*?s'

p_greedy = re.compile(pattern_greedy)
p_nongreedy = re.compile(pattern_nongreedy)

word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'

m_greedy = p_greedy.search(word)
m_nongreedy = p_nongreedy.search(word)

print(m_greedy)
print(m_nongreedy)
```

```
>>>
<re.Match object; span=(0, 45),
match='Pneumonoultramicroscopicsilicovolcanoconiosis'>
<re.Match object; span=(0, 19), match='Pneumonoultramicros'>
```

Groups

Suppose we want to capture individual components (i.e. groups) of a pattern. For example, let's imagine that a particular identifier comprises a word, then a numerical value and then another word. Well we can do this by creating groups within the regex using round brackets. See the code below:

```
import re

pattern= '^([A-z]+)(\d+)([A-z]+)$'
p = re.compile(pattern)
seq = 'The6Hundred'

m = p.search(seq)

print(m)
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(3))

>>>
<re.Match object; span=(0, 11), match='The6Hundred'>
The6Hundred
The
6
```

We have developed a more complex regex than encountered before, but once you break it down, it is quite intelligible. The first component of our serial are letters, as represented by the character class `[A-z]`. Since there will be at least one of these letters, we suffix the character class with a +. Then we have 1 or more digits, represented by `\d+`. Finally, we end with `[A-z]+` once more, representing another word. Since there should be nothing else within our serial number we ensure that only this pattern was retrieved (i.e. there is nothing either side) by delimiting the regex with `^` and `$`. We now define our groups by surrounding `[A-z]+` and `\d+` with rounds brackets.

To retrieve the value associated with each of the three groups, use the group method of the match object. The value at position 0 will be the whole matched term (i.e. `The6Hundred`), while values 1-3 correspond to each of the predefined groups, proceeding in order, from left to right of the regex.

Compilation flags

Compilation flags allow you to modify how certain aspects of a regular expression works. The more commonly used flags are `IGNORECASE` (`I`) and `MULTILINE` (`M`). The former causes the regular expression to perform case-insensitive matches on text characters. The latter flag modifies the mode of action of the metacharaters `^` and `$`. The caret (`^`) now matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. In a similar fashion, the `$` metacharacter matches either at the end of a string and at the end of each line. Add the appropriate flag when creating the regular expression, for example:

```
import re

pattern= 'AAGCTT'
p = re.compile(pattern)
p_ignore_case = re.compile(pattern, re.IGNORECASE)

seq = 'aagctt'

m = p.search(seq)
m_ignore_case = p_ignore_case.search(seq)

print(m)
print(m_ignore_case)

>>> %Run regext.py
None
<re.Match object; span=(0, 6), match='aagctt'>
```

You can see how applying the `IGNORECASE` has modified the action of the regular expression, causing `aagctt` to match `AAGCTT`.

Modifying Strings

Regular expressions can also be used to modify strings in a variety of ways. Perhaps the two most commonly used methods to perform such a modification are **`split()`** and **`sub()`.**

The `split()` method splits a string into a list, subdividing it wherever the regex matches. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. The regex may also be given a `maxsplit` value, which limits the number of components returned by the splitting process.

Another task is to find all the matches for a pattern, and replace them with a different string. The **`sub()`** method takes a replacement value – which can be either a string or a function – and the string to be processed. Similar to the `split()` method, an optional argument count may be passed in the regex specifying the maximum number of pattern occurrences to be replaced.

If you look at the example below you will see code that modifies the last word in our serial ID (from before) into `Thousand`. The regex identifies the last word in the string. We then perform the `sub()`

method on the pattern object, passing the replacement value as well. This will replace `Hundred` to `Thousand` in the string.

```
import re

pattern= '[A-z]+$'
p = re.compile(pattern)
seq = 'The6Hundred'

m = p.sub('Thousand', seq)

print(m)

>>>
The6Thousand
```

That brings this chapter on regular expressions to an end. Although we have covered a wide range of scenarios, the symbolic representation of patterns makes regular expressions a language in its own right. Consequently, if you find at some point in the future that what has been mentioned here doesn't provide you with the search term you need, you should certainly refer to the Python documentation, from where you will receive even more information on regular expression construction.

Concluding remarks

Well, that brings the Advanced Python course to an end. You should now have at your disposal a wide range of computational tools that can be employed to construct simple scripts, develop software or perform some kind of data analysis. Not only can you write code, but you should be able to write well-structured code that is intelligible to other people. You should now understand how to handle errors elegantly or import modules to perform a wide range of tasks simply and quickly. In addition, you can now undertake pattern matching – a key area in nucleic acid or protein sequence analysis – by writing regular expressions.

As discussed in our previous Python cause, we strongly recommend that you go out of your way to find reasons to write code over the next few weeks. If you don't build upon your current knowledge, as the weeks turn into months, you will become less familiar with what you have learned over the past few days. Maybe there is some analysis that you could now perform with Python? Even if it is easier to do the tasks in, say, MS Excel, reinforcing your new-found skills now will pay dividends in the future.

Learning Python is akin to learning a foreign language. There is a great deal to take in and becoming fluent takes practice, practice, practice.

We would like to bring to your attention the following resources that may help you in your future Python career:

www.python.org – the homepage of Python. This should often be your first port of call for Python-related queries.

www.jupyter.org – many bioinformaticians and computational biologist are adopting Jupyter notebooks to write code and share results in a structured and reproducible fashion.

www.matplotlib.org – a popular resource for using Python to produce graphs and charts.

pandas.pydata.org – an open source data analysis and manipulation tool.

www.biopython.org – a set of freely available tools for biological computation.

Also, don't forget the Babraham Bioinformatics pages listing available courses and providing training materials: https://www.bioinformatics.babraham.ac.uk/training

Our **Data Analysis in Python** and **Object Oriented Programming in Python** courses may be of particular interest to you.

Happy coding!
The Babraham Bioinformatics Team