Advanced Python Course

Steven Wingett, Babraham Bioinformatics

version 2020-08

Babraham Bioinformatics

Licence

Writing better code

Part 1

Babraham Bioinformatics

Comprehensions

- Elegant way to achieve same tasks as using loops, but in a single line
- Used to create sets, lists or dictionaries from a collection

Babraham Bioinformatics

Comprehensions (2)

- **List comprehensions** – generate list as output
- Syntax: `[expression for item in collection]`
- Example, the square numbers from 0 to 100:

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

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

Comprehensions (3)

- **Set comprehensions** – generate set as output

- Syntax: `{expression for item in collection}`

- Sets only contain unique items, so you cannot be sure that the output will have the same number of elements as the input

```
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'}
```

Comprehensions (4)

- **Dictionary comprehensions** – generate a dictionary as output
- Keys and values to contend with
- Syntax:

  ```
  {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}
```

Babraham Bioinformatics

Comprehensions (5)

- **Conditional comprehensions** filter collections

- Syntax:
  `[expression for element in collection if test]`

- Filter for numbers less than 3:

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

Babraham Bioinformatics

Exercises

- ***Exercise 1.1 – Comprehensions***

Ternary Expressions – simplify code

Bloc sytax

```
x = 2
if x > 1:
    value = "Yes"
else:
    value = "No"
print(value)

>>>>
Yes
```

Ternary expression

```
x = 2
value = "Yes" if x > 1 else "No"
print(value)

>>>
Yes
```

Babraham
Bioinformatics

Complex data structures

```
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'>
```

Babraham Bioinformatics

The zip() function

- groups elements of lists, tuples and other collections to form a list of tuples
- returns a zip object, which is an iterator of the combined tuples
- list() function then creates a list of tuples
- the collection with the **fewest elements** decides the length of the zip iterable

```
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')]
```

Exercises

- ***Exercise 1.2 – Complex data structures***

A note on scope

- A name need not be accessible to every part of a script – known as **scoping**

Note 1: variables created in an outer bloc of code will be accessible to the inner bloc of code

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

>>>
10
11
```

Babraham Bioinformatics

A note on scope (2)

- **Note 2:** similarly, variables declared outside a function are available within that function

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

x = 1
test_function()

>>>
1
```

Babraham Bioinformatics

A note on scope (3)

Note 3: Loops can modify variables "external" variables, but functions cannot

Loop

```
x = 1

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

>>> %Run mypy.py
3
```

Function

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

x = 1
test_function()

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

A note on scope (4)

Note 4: Variables created within a function will not be accessible outside that function

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

print(add_one(1))
print(x)

>>>
2
Traceback (most recent call
last):
…
```

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

print(add_one(1))
print(y)

2
Traceback (most recent call
last):
…
```

A note on scope (5)

Note 5: variables created inside a loop will be accessible outside that loop

Code

```
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))
```

Result

```
>>>
0
10
10

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

Exercises

Scope: exercises 1.3 and 1.4

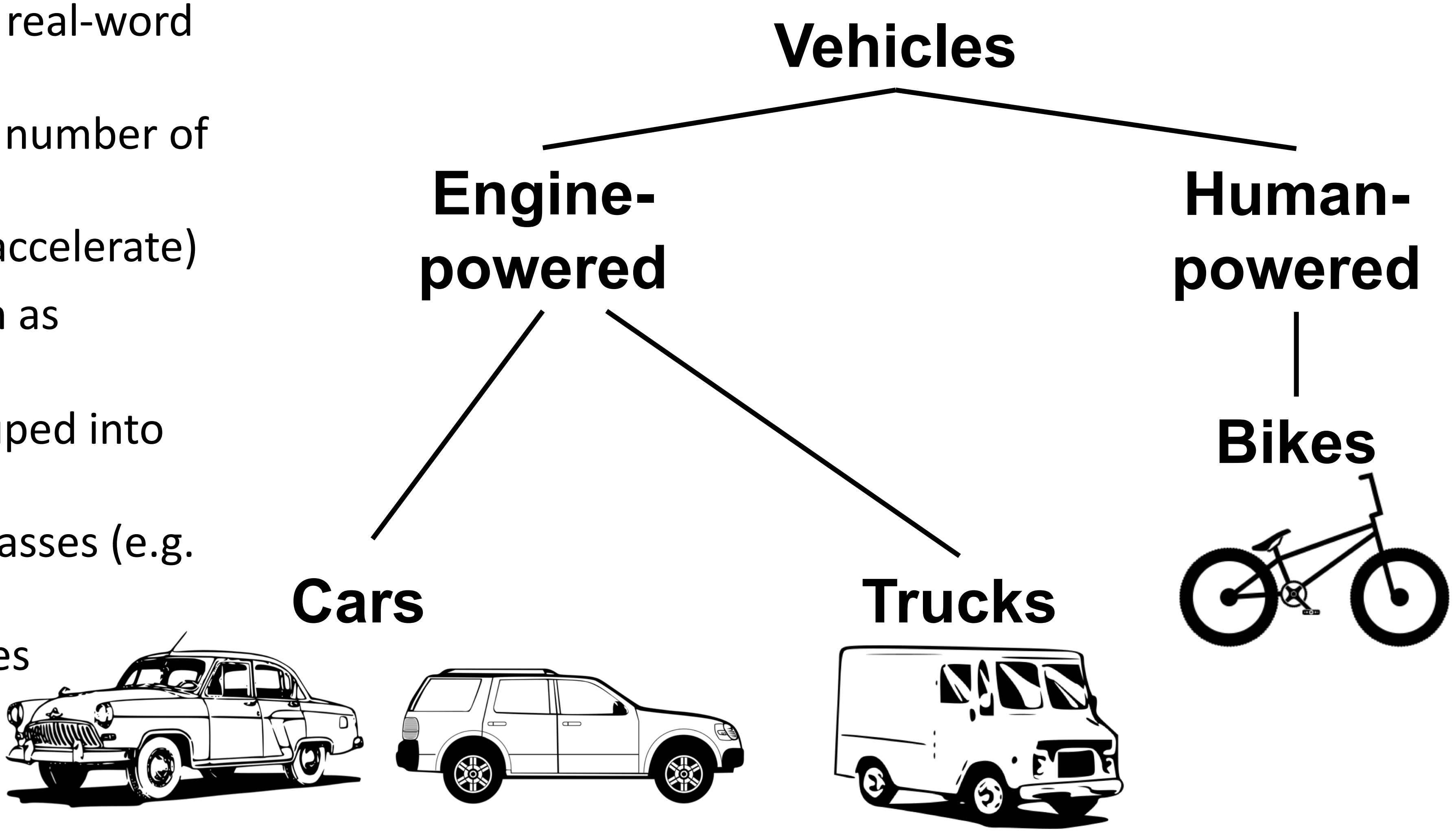
Introduction to object-oriented programming

- So far dealt with Python as a **procedural** language – a series of instructions (like a food recipe)
- Easy to loose track of everything for big projects
- Object-oriented programming (OOP) designed to make it easier to writing more complex projects
- It is better suited to the human brain
- In fact, everything in Python is an object

Babraham Bioinformatics

Introduction to object-oriented programming (2)

- Object are analogous to real-word objects (e.g. vehicles)
- Objects have fields (e.g. number of wheels, max speed)
- And methods (turn on, accelerate)
- Methods & fields known as attributes
- Related objects are grouped into classes (i.e. vehicles)
- And grouped into sub-classes (e.g. cars, trucks and bikes)
- Objects inherit properties



Babraham Bioinformatics

Introduction to object-oriented programming (2)

Class Code

```
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')
```

“Main body” code

```
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())
```

Output

```
Happy
Sad
Angry
Bite
```

Babraham Bioinformatics

Exercises

- Object orientated programming: Exercise 1.5

Generators

- As previously noted, generators produce a stream of data
- Often preferable to keeping a very large list in memory, and then selecting a value from the list when required
- Syntax almost identical to that of a function
- **Except**, the generator code uses the keyword `yield` instead of `return`
- `next` instigates each iteration of the generator
- Generators keep track of “where they have got to”
- Can (in theory) produce an infinite number of values
- Alternatively, we can specify limit after which `StopIteration` error occurs

Babraham
Bioinformatics

Generators (2)

Code

```
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))
```

Result

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

- Like functions, generators can take arguments:

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

Generators

- Exercises: 1.6, 1.7, 1.8
- Optional: 1.9*, 1.10*

Exception handling: how to handle errors

- You will have seen Traceback messages by now
- These occur when there are errors in your code
- Sometimes these are because your code cannot be interpreted (syntactically incorrect) – solution: re-write your code
- Sometimes there is no way to re-write code to avoid errors
- Errors detected during runtime are called **exceptions**
- We can handle them exceptions and make our program “fail gracefully”
- Unhandled exceptions will result in a traceback message

How to handle errors (2)

- Use the `try` statement followed by the `except` clause to handle errors

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

How to handle errors (3)

- Look at this error:

```
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
```

How to handle errors (4)

- A possible solution is to add a `try` statement:

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


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

- the code does not terminate with a traceback, but prints to the screen a user-friendly description of what went wrong
- if this code were part of a larger program, the program should carry on running

How to handle errors (5)

- Error handling may involve multiple considerations, as illustrated in the template below:

```
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
```

Babraham Bioinformatics

Error handling

- Exercises 1.11 and 1.12
- Optional 1.13*

Modules

Part 2

Babraham Bioinformatics

Modules

- Modules make available to the user a much greater range of data types, functions, and methods
- Many in-built modules and many external modules
- Modules are files placed in a library directory during Python installation
- To import `os` module:

```
import os
```

- Reference module when accessing newly imported name or method:

```
os.getcwd()
```

- To import specific names only from a module:

```
from os import getcwd
```

- Now you can refer to `getcwd()` directly (no need of `os`)

Getting used to modules

- So now let’s look at some of the modules that you will most likely find useful
- When you want to perform a task with an additional module, there is often some degree of research involved
- Does my desired module actually exist? Check existing modules
- Check the Python documentation and online to find a module that appears suitable
- Reading the module’s documentation
- Incorporate module into codebase
- The exercises for this chapter were designed to help you go through some of this process yourself

Module: `datetime`

- For handling date / time data
- Online documentation for modules: https://docs.python.org/3/library/datetime.html

- Quite technical, but you should be able to understand some of it and this will get easier to understand

Babraham Bioinformatics

Module: `datetime`

Extract of online documentation on `datetime`

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.

Module: `datetime`: usage introduction

- Import datetime module, then instantiate a datetime object:

```
import datetime

my_datetime_object = datetime.datetime.now()

print(my_datetime_object)

>>>

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

As an aside, the keyword `dir()` lists module attributes:

```
print(dir(datetime))

>>>

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

Module: `datetime:` usage introduction (2)

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
```

Can be used to add or subtract dates

```
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
```

Module: `math`

- For mathematics beyond that available in standard Python
- Provides trigonometric & hyperbolic functions, powers and logarithms etc.
- Includes mathematical constants such as pi and e
- Example (logs):

```
import math

print(math.log10(1000))

>>>
3.0
```

Babraham Bioinformatics

Module: `sys`

- Provides information about the Python implementation being used
- `sys.argv`: a list of command line arguments used to run the script (the first element is the name of the script itself)
- `sys.modules`: dictionary in which the keys name the currently loaded modules
- `sys.path`: list of of the directories into which Python looks for modules following the import command in a script:

```
['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():` function that causes a Python program to exit. By convention, pass the argument 0 if the program exits without error, else pass another integer value

Module: `time`

- Most likely to use the function: `time.sleep()` from this module
- Causes the program to wait for the specified number of seconds passed as an argument to the function
- Slowing down a program is a useful way to check output written to the screen at a pace that is readable by a human
- Sometimes we may add a delay to a script if it is part of a pipeline

Babraham Bioinformatics

Exercises

- Exercises 2.1, 2.2, 2.3 and 2.4

Module: `subprocess`

- Command line introduction and demo
- Allows a Python script to interact and manipulate your system (e.g. launch another script as part of a pipeline)
- You can now write shell commands into your Python scripts
- Example:

```
import subprocess

print(subprocess.getoutput('echo "Hello World!"'))
---
user$ python3 subprocess_example.py
Hello World!
```

Babraham Bioinformatics

Module: `os`

- An operating system interface
- Most commonly used in Python scripts for interacting with files and directories in the filesystem
- Examples:

  `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

Module: `tempfile`

- A Python script may, during processing, write out a temporary file
- What happens if you are running multiple instances of the same script? Will the temporary file be overwritten?
- Not a trivial problem to solve, but `tempfile` does it for you

Module: `glob`

- Suppose there is a file or a list of files on your computer that needs processing but you don’t know the exact names of the files in advance
- `glob` provides enables your script to search the filesystem, return the relevant filenames and process these
- Usage: `glob.glob(`*`pattern`*`)`

Wildcard	Explanation
*	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

Module: `glob` (2)

```
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']
```

Babraham Bioinformatics

Module: `string`

- The **string module** (not the `str` data type) contains a range of useful values and functions

- For example, it contains all the upper- and lowercase letters in:
  `string.ascii_uppercase`
  `string.ascii_lowercase`

- Also names representing digits, punctuation and a range of other string values

Exercises

- Exercises 2.5, 2.6, 2.7 and 2.8

Module: `csv`

- Data files often use **comma-separated values (CSV)**
- Stores a table or matrix (with layout similar to that of *MS Excel*), only it uses commas to separate columns.
- Uses newline character to separate rows
- Similarly, **tab-separated values** (or **tab-delimited**) format uses tabs to separate columns
- The **csv module** enables a Python script to read from or write to these types of file.

Modules: `zlib` and `gzip`

- Modern life science data files are often are very large
- So may be compressed as either zip or gzip files
- `zlib` and `gzip` modules enable reading from or writing to such files
- Syntax similar to that of opening a regular file
- One important difference: returned data a sequence of bytes (note a `b` before the quotation mark when printing to the screen)
- 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

Modules: `zlib and gzip` (2)

Code

```
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)
```

Output

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

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

Babraham Bioinformatics

Module: `textwrap`

- For formatting text in a consistent, neat and logical manner
- A lengthy string may be converted into a list by `textwrap.wrap`
- And then 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

Module: `textwrap` (2)

```
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))
```

Babraham Bioinformatics

Module: `textwrap` (3)

One long string................................

```
>>> %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.
.
.
.
```

Exercise

- Exercises 2.9, 2.10 and 2.11
- Optional 2.12* and 2.13*

Module: `argparse`

- Makes it possible to modify the action of a Python script via the command line:

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

- Enables arguments to be passed
- Enables flags
- Flags with additional arguments (and specifing whether these should be integers or strings)
- Automatically generates a --help option

Babraham Bioinformatics

Exercises

- Exercise 2.14

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 (e.g. in bioinformatics) that require downloading and installation

- If all goes well, this process should be quite simple thanks to the **`pip`** installer, which is included by default with Python since version 3.4

Babraham Bioinformatics

Installing Modules and Packages (2)

- Install command:

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

- Specific version:

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

- At least as recent as:

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

- 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
```

Installing Modules and Packages (3)

- Find out where are Python packages installed with command:

```
python3 -m site
```

- Returns text similar to:

```
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`

Installing Modules and Packages (4)

- If you do not have administration rights to write to the `site-packages` folder, you should be able to install packages that are isolated just 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

- If `pip` does not work, the module required should give instructions on to how it should be installed

Virtual Environments

- Allows Python packages to be installed in an isolated location for a particular application, rather than being installed globally
- 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

Babraham Bioinformatics

Biopython

- **Biopython** is a set of freely available Python tools for biological computation
- The Biopython homepage is found at: https://biopython.org/
- To install:

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

Biopython (2)

- Example to parse a Genbank format file to a FASTA file:

```
from Bio import SeqIO

with open("cor6_6.gb", "rU") 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")
```

- Very efficient, saving you time and writing code
- Reduces the chance of mistakes, since many people use this codebase

Exercises

- Exercise 2.15

Regular expressions

Part 3

Babraham Bioinformatics

Introducing regular expressions

- **Regular expressions** (**regexes** or **RE**s) enable advanced pattern matching of strings
- Very useful in a range a situations e.g. matching nucleic acid sequences
- Almost a programming language in their own right!
- Once you have learnt for Python, you should be able to use them in Java, Perl etc.
- Import the `re` module which provides an interface to the regular expression engine

Simple regex pattern matching (DNA/HindIII)

Code

```
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))
```

Make pattern object

Return match object

Output

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

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**

Babraham Bioinformatics

Querying the match object

Code

```
import re

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

seq = 'NNAAGCTT'

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

Output

```
>>>
AAGCTT
2
8
```

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

Exercises

- Exercise 3.1

Metacharacters

- What was the point of being able to print the matched sequence, if we already know what it is?
- Metacharacters allow is to create patterns that represent a vast range of possible strings (which we may not know in advance)
- The list of regex metacharacters is:
  `. ^ $ * + ? { } [ ] \ | ( )`

Babraham Bioinformatics

Metacharacters: Character classes

- metacharacters **[** and **]** are used to define classes of characters
- [abc] means match the letters a **or** b **or** c
- [a-c] will also achieve the goal
- Similarly [a-z] matches all lowercase characters
- Metacharacters generally lose their special properties inside the square brackets
- Caret complements a set: [^7] will match any character *except* 7

Metacharacters: start and ends

- We are now referring to characters placed **outside** of the character class
- caret (**^**) denotes the start of the string
- dollar symbol ($) denotes the end of a string
- the regex ^ABC$ would mean a matching pattern should contain ABC and have nothing either side
- ^pi would match pi, pike, and pill, but not spill

Babraham Bioinformatics

Metacharacters: dot (.)

- Matches **any** character, except newline characters

Babraham Bioinformatics

Metacharacters: pipe (|)

- The pipe metacharacter (|) means ‘or’ in a regex
- a|b is the same as [ab])

Babraham
Bioinformatics

Metacharacters: backslash

- Backslash (\) serves multiple purposes in regexes
- Often followed by a letter character, and together those two characters represent a whole range of different characters

\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_]

Babraham Bioinformatics

Metacharacters: backslash (2)

- You may have notices problems with our rules e.g. dot is a metacharacter, so how do we represent a dot in a regex
- Answer, escape the dot: `\.`
- Preceding a metacharacter with backslash will cause it to be interpreted literally
- You can also escape the backslash character itself:

```
pattern = '\\'
```

Babraham Bioinformatics

Exercises

- Exercise 3.2

Raw String Notation

- Alternative to escaping characters (which may become very difficult to read) is to use **raw string notation**
- 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"`

Babraham Bioinformatics

Repetition

- We can specify phrases that need to be repeated a specified number of times for successful match

- This uses metacharacters:

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

Babraham Bioinformatics

Repetition (2)

Code

```
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)
```

Output

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

Repetition (3)

- Repeat characters: `a{n}`

- Another method: place two integers between curly brackets: `{m,n}`

- There must be at least `m` repetitions, and at most `n` of the preceding character

- For example: `a/{1,3}b`
  - matches: `a/b`
  - matches: `a//b`
  - matches `a///b`
  - no match: `ab`
  - no match: `a////b`

- Omitting the `m` is interpreted as a lower limit of 0 `{ , n}`, while omitting *n* results in an upper bound of infinity `{m, }`

Greedy vs non-greedy matching

Code

```
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)
```

Output

```
>>>

<re.Match object; span=(0, 45),
match='Pneumonoultramicroscopicsilicovolcanoconios
is'>

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

- Greedy matching matches as many characters as possible, while non-greedy matching matches the fewest allowed characters
- We achieve non-greedy matching by appending `?` to the repetition metacharacters:
  `*?     +?     ??     {m,n}?`

Metacharacters: groups

- Suppose we want to capture individual components (i.e. groups) of a pattern
- Define groups using round brackets: `()`
- e.g. an identifier comprises a word, then a numerical value and then another word
- Index 0 returns the whole match object, the remaining index values returns the respective groups

Code

```
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))
```

Output

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

- Use the `group()` method of the match object to retrieve the value associated with each of the three groups

Compilation flags

Code

```
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)
```

Output

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

Compilation flags modify regular expressions

- IGNORECASE (I) causes the regular expression to perform case-insensitive matches
- MULTILINE (M) modifies the mode of action of the metacharaters ^ and $. The caret (^) now matches at i) the beginning of the string and ii) the beginning of each line within the string; $ now matches either i) at the end of a string or ii) at the end of each line

Modifying strings

- Regular expressions can modify strings in a variety of ways
- Two most commonly used methods are **`split()`** and **`sub()`**
- `split()` subdivides a string into a list 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
- `sub()` method takes a replacement string value and the string to be processed
    - The regex may also be given a `maxsplit` value

Modifying strings (2)

```
import re

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

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

print(m)

>>>
The6Thousand
```

Exercises

- Exercise 3.3

Exercises

- Exercise 3.4

Babraham Bioinformatics

Concluding remarks

- We have covered a lot of material !
- You should now be familiar with the Python datatypes, understand concepts such as functions and methods and how programs are controlled using loops and conditional operators
- We have also introduced object orientated programming
- We strongly recommend that you go out of your way to find reasons to write code over the next few weeks – else you will forget!
- Learning Python is akin to learning a foreign language

Babraham Bioinformatics

Resources

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.

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

Babraham Bioinformatics

How do you get to Carnegie Hall? Practice, practice, practice.

Happy coding!

The Babraham Bioinformatics Team

Babraham Bioinformatics