Advanced Python

Ade Fewings, Aaron Owen
Bangor University
Updated: 2022-02-16

eResearch

- Supercomputing Wales
  - Available to researchers at Bangor
- Research Software Engineers
- Collate expert knowledge into an open and shared centralised repository
  - Yammer
  - Github
  - Workshops
  - Projects
  - Acknowledgements

Training Workshops

- Introduction to the Linux Shell
- Version Control Using Git
- Programming Principles and Practice using Python
- **Advanced Python**
- Parallel Processing in Python
- Machine Learning with Python*

See and discuss on the Yammer group.

Suggestions for new training welcome.

Environment

Cross Platform

- Google Colaboratory
- Online Python compiler, Online Python IDE, and online Python REPL
- python.org shell

Python3 at Bangor - Recommend

- Visit https://jupyter.bangor.ac.uk/jupyter/hub/login
- Login using your Bangor University credentials.
- Select the `Python 3.8` notebook.

Formatted strings

% Operator

```
name='Aaron'
print('Hello %s %d' % (name, 32))
```

str.format

```
name='Aaron'
print('Hello {n} {a}'.format(
  n=name,
  a=32
))
```

F Strings (Python 3.6+)

```
name='Aaron'
age=32
print(f'Hello {name} {age}')
```

```
a=5
b=10
print(f'{2*(a+b)}')
```

Multiple Function Arguments

- Every function in Python receives a predefined number of arguments if declared normally, like this:

```
def test_function(first, second, third):
  print(first, second, third)

test_function('a', 'b', 'c')

> a b c
```

Multiple Function Arguments

- It is possible to declare functions which receive a variable number of arguments, using the following syntax:

```
def foo(first, second, third, *the_rest):
  print(f'First: {first}, Second: {second}, Third: {third}')
  print(f'And all the rest as a tuple ... {the_rest}')
  print(f'And all the rest as a list ... {list(the_rest)}')

foo('a', 'b', 'c', 'd', 'e')
```

```
> First: a,Second: b, Third: c
> And all the rest as a tuple ... ('d', 'e')
> And all the rest as a list ... ['d', 'e']
```

- The `the_rest` variable is a list of variables, which receives all arguments which were given to the `foo` function after the first 3 arguments.

Multiple Function Arguments

```
def foo(a, b, c, *args):
    print(args)

foo(1,2,3,4,5,6)

> (4,5,6)
```

```
def bar(a, b, c, **kwargs):
    print(kwargs)

bar(1,2,3,name='Aaron',age=31)

> {'name': 'Aaron', 'age': 31}
```

Multiple Function Arguments

- It is also possible to send functions arguments by keyword.

```
def foo(first, second, third, **options):

    if options.get("action") == "sum":
        print(f"The sum is: {first + second + third}")

    if options.get("number") == "first":
        print(f"The first number is {first}")

foo(1, 2, 3, action = "sum", number = "first")
```

```
> The sum is: 6
> The first number is 1
```

- The `foo` function receives 3 arguments.
- If an additional `action` argument is received, and it instructs on summing up the numbers, then the sum is printed out.
- If an additional `number` argument is received, and it instructs on printing the first argument, then the first argument is printed out.

Exception Handling

- Python's solutions to errors are exceptions.
- Traceback?

```
print(a)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

- Often you do not want exceptions to completely stop the program.
- If a network service went down, you would not want your program to fail and permanently stop.
- We can **catch** errors by using the `try/except` block.
- Most languages provide a mechanism to throw and catch errors.

Exception Handling

- Let's catch an `IndexError` when trying to access items that are not in a list.

```
try:
  names=['Aaron', 'Beth', 'George']
  print(names[4]) # Should throw an error
except IndexError:
  # Enters this section when an IndexError is caught
  print('An IndexError was caught')
print('We can carry on, the program did not stop')
```

```
> An IndexError was thrown
> We can carry on, the program did not stop
```

Exception Handling

- Let's catch a `NameError` when trying to access a variable that does not exist.

```
try:
  print(x)
except NameError:
  # Enters this section when an NameError is caught
  print('An NameError was caught')
print('We can carry on, the program did not stop')
```

```
> An NameError was caught
> We can carry on, the program did not stop
```

Exception Handling

- Often an error is thrown but we do not know what kind of error to catch.

```
try:
  print(x)
except Exception:
  # Enters this section when an Exception is caught
  print('An Exception was caught')
print('We can carry on, the program did not stop')
```

```
> An Exception was caught
> We can carry on, the program did not stop
```

- The `Exception` class is the top-level error class.
- All other errors derive from the `Exception` class.

Exception Handling

- You can use the `try/except` syntax for any exception in Python.
- https://docs.python.org/tutorial/errors.html#handling-exceptions
- It is possible to raise an exception manually.

```
try:
  # Some logic that I do not agree with
  raise Exception('Custom Exception message')
except Exception as e:
  print(e)
> Custom Exception message
```

Exception Handling

- The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  # Some logic that I do not agree with
  raise Exception('Custom Exception message')
except Exception as e:
  print(e)
finally:
  print("The 'try except' is finished")
```

```
> Custom Exception message
> The 'try except' is finished
```

- Useful for file IO

Exception Handling

- The `try` block lets you test a block of code for errors.
- The `except` block lets you handle the error.
- The `finally` block lets you execute code, regardless of the result of the `try` and `except` blocks.

Classes and Objects

- Python is an object-oriented programming language.
- Almost everything in Python is an object, with its properties and functions.
- Objects are an encapsulation of variables and functions into a single entity.
- Objects get their variables and functions from classes.
- Classes are essentially a template to create your objects (Blueprint - House).
- To create a class use the `class` keyword.

```
class MyClass:
  x=5
```

- Can use the class named `MyClass` to create objects

```
obj = MyClass()
print(obj.x)
```

The init Function

- The previous slide demonstrated a very simple class.
- All classes have a function called `__init__()`, which is always executed automatically when the class is being initiated.

```
class Test:
    def __init__(self):
        print("Test's __init__ func()")

obj = Test()
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Aaron", 31)
print(p1.name)
print(p1.age)

p2 = Person("Beth", 29)
print(p2.name)
print(p2.age)
```

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def my_func(self):
    print("Hello my name is " + self.name)

p1 = Person("Aaron", 31)
p1.my_func()

print(dir(p1)) # Very advanced
```

The self Parameter

- The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.
- It does not have to be named `self`, you can call it whatever you like, but it has to be the first parameter of any function in the class:

```
class Person:
  def __init__(s, name, age):
    s.name = name
    s.age = age

  def my_func(s):
    print("Hello my name is " + s.name)

p1 = Person("Aaron", 31)
p1.my_func()
```

- Prefer `self`.

Modify Object Properties

- You can modify properties on objects

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Aaron", 31)

print(p1.name)

p1.name="Beth"

print(p1.name)
```

Store a collection of Objects

- Create multiple objects and store them in a list.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Aaron", 32)
p2 = Person("Beth", 28)
p3 = Person("George", 2)

people = [p1,p2,p3]

for person in people:
    print(f'{person.name} is {person.age} years old')
```

Modules

- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.
- To create a module save the code in a file with the extension `.py`.
  - Save this code to a file named `hello.py`

```
def greeting(name):
  print(f"Hello {name}")
```

  - Save this code to a file named `main.py`

```
import hello

hello.greeting("Aaron")
hello.greeting("Beth")
```

Modules

- When using a function from a module, use the syntax: `module_name.function_name`
- The module can contain functions, as already described, but also variables of all types (lists, dictionaries, objects etc)
- Developers often place configuration files in seperate modules.
  - Save this code to `people.py`

```
person_1 = {"name": "Aaron", "age": 31}
person_2 = {"name": "Beth", "age": 29}
```

  - Save this code to `main.py`

```
import people

print(people.person_1)
print(people.person_2)
```

Importing Classes

- Best practice is to store the class definition in a separate file and import the class when required.

Person.py

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

main.py

```
from Person import Person

p1 = Person("Aaron", 32)
p2 = Person("Beth", 28)
p3 = Person("George", 2)

people = [p1,p2,p3]

for person in people:
    print(f'{person.name} is {person.age} years old')
```

Importing Functions

- Best practice is to store the function definition in a separate file and import the function when required.

util.py

```
def add(x,y):
    print(x+y)

def divide(x,y):
    print(x/y)
```

main.py

```
from util import add, divide

add(5,6)
divide(5,2)
```

Modules (Re-naming)

- You can create an alias when you import a module, by using the `as` keyword:

```
import people as p

print(p.person_1)
print(p.person_2)
```

- Very common to abbreviate modules
  - `import numpy as np`
  - `import pandas as pd`
  - Developers like to type less.

Modules (Built-In)

- There are several built-in modules in Python, which you can import whenever you like.

```
import platform

print(platform.system())
print(platform.node())
```

```
import os

print(os.getcwd())
```

- https://docs.python.org/3/py-modindex.html

Assertions

- Fall into three categories
  - Precondition
    - A condition must be true at the start of a function for it to work correctly. Example, checking input args.
  - Postcondition
    - A condition that the function guarantees is true when it finishes. Example, checking result is valid.
  - Invariant
    - A condition that is always true at a particular point inside a piece of code. Example, age can't be negative.
- Extremely useful to ensure your object maintains a valid state (age?).

Pre and Post Conditions Demo

- Suppose you are writing a function called average that calculates the average numbers in a list. What pre and post conditions would you check for?

```
# A possible pre-condition:
assert len(input_list) > 0, 'List length must be non-zero'
```

```
# A possible post-condition:
assert min(input_list) ⩽ average ⩽ max(input_list),
'Average should be between min and max of input values (inclusive)'
```

Assertions

- They are not just about catching errors.
- They help people to understand your code - documentation.
- Whenever you fix a bug or error in your code, write an assertion that catches the mistake.

Lambda functions

- A lambda function is a small anonymous function - anonyomus in the sense that it does not actually have a name.
- Python functions are typically defined using `def function_name()`.
- Lambda functions can take any number of arguments, but must always have one expression.

```
x = lambda a, b : a * b
print(x(5, 6))
```

- A lambda function that sums argument a, b, and c and print the result

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

- A lambda expression is a block of code that can be passed around to execute. A possible use case is to define filtering logic as the first argument for `filter()`

Map

- `map()` is a built in Python function that is used to apply a function to a sequence of elements.
- It offers a very clean and elegant way to express an idea and most importantly it improves the readability of your code.

```
def func(i):
  return i*i

data = [1,10,100]

result = map(func, data)

print(list(result))
```

- It's more pythonic to use list comprehensions.

Filter

- `filter()` is a built in Python function that is very similar to `map()` however, `filter()` will only return the elements which the applied function returned as True.

```
def remove_odd_numbers(num):
    if num % 2 == 0:
        return True
    else:
        return False


numbers = range(15)
filtered_numbers = filter(remove_odd_numbers, numbers)
print(list(filtered_numbers))
```

List Comprehensions

- List Comprehensions are a very powerful tool.
- Creates a new list based on another list, in a single, readable line.
- For example, let's say we need to create a list of integers which specify the length of each word in a certain sentence.

```
sentence = "the quick brown fox jumps over the lazy dog"
words = sentence.split()
word_lengths = [len(word) for word in words]
print(words)
print(word_lengths)
```

Exercise

- Using a list comprehension, create a new list called `new_list` out of the list `numbers`, which contains only the positive numbers from the list, as integers.

```
numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]
new_list = []  # Some logic
print(new_list)

> [34, 44, 68, 44, 12]
```

- Solution

```
numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]
newlist = [x for x in numbers if x > 0]
print(newlist)

> [34, 44, 68, 44, 12]
```

- List comprehension is popular and people like to display their skills. Simply break the commands down and process them step by step.

Itertools

- Python provides a module called *Itertools* that is a collection of tools for handling iterators.
- An iterator is a data type that can be used to loop thorugh a collection of data items such lists, tuples and dictionaries.

```
import itertools

# Join two lists into a list of tuples
for i in zip([1, 2, 3], ['a', 'b', 'c']):
    print(i)

# Group data values
data = sorted([1, 2, 1, 3, 2, 1, 2, 3, 4, 5])
for key, value in itertools.groupby(data):
    print(key, list(value))

print(list(itertools.permutations('AB')))
print(list(itertools.accumulate([1, 2, 3, 4])))
print(list(itertools.chain([1, 2], [3, 4])))
```

Generators

- Generator functions allow you to declare functions that behave similar to an iterator but can be much more efficient.
- Consider calculating the sum of numbers from 1 to 1000 using a list and a for loop. This could require 1000 integer values to be stored in memory. Small memory usage.
- Consider calculating the sum of numbers from 1 to 1,000,000,000. This would create a huge list in memory, especially if the data items were floats.
- The solution is to use a generator that will create elements and store them in memory only as it needs (one at a time).

```
def generate_numbers(n):
    num = 0
    while num < n:
        yield num
        num += 1

total = sum(generate_numbers(100000))
print(total)
```

Decorators

- Decorators allows programmers to modify the behaviour of a function or class.
- Wrap a function with extended logic without modifying the original code. Sometimes the source code to the original code might not be available.
- Everything in Python is an object, even functions.
- Functions can be stored as a variable, passed as a parameter to another function, returned from another function and stored in data structures.

```
# Stored as a variable
def func(a):
  print(a)

func('Hello')

x = func
x('Hello')
```

Decorators

```
# Passed as a parameter to another function

def shout(text):
    return text.upper()


def whisper(text):
    return text.lower()


def greet(func):
    msg = func(f'I was created by {func.__name__}')
    print(msg)

greet(shout)
greet(whisper)
```

Decorators

```
# Returned from another function

def create_adder(x):
    def adder(y):
        return x + y

    return adder

add_15 = create_adder(15)

print(add_15(10))
print(add_15(100))
```

Decorators

```
def test_decorator(func):
    def wrapper(*args, **kwargs):
        print(f'Before {func.__name__} is called')
        result = func(*args, **kwargs)
        print(f'After {func.__name__} is called')
        return result

    return wrapper


@test_decorator
def add_func(a, b):
    print(f'Inside add()')
    return a + b


a, b = 2, 5
print(add_func(a, b))
```

Getters and Setters

- Avoid `get_` and `set_` methods, not pythonic.
- Use the property decorator.

```
class Celsius:
    def __init__(self, temperature = 0):
        self._temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        #print("Getting value")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        #print("Setting value")
        self._temperature = value
```

Getter and Setters

```
c = Celsius()

print(c.temperature)           # 0
print(c.to_fahrenheit())       # 32

c.temperature = 30
print(c.to_fahrenheit())       # 86

c.temperature = -300
```

- Need to add validation to `__init__` method?

File Handling

- Python has several functions for creating, reading, updating, and deleting files.
- The key function for working with files in Python is the `open()` function.
- The `open()` function takes two parameters; filename, and mode.
- There are four different methods (modes) for opening a file:
  - "r" - Read - Default value. Opens a file for reading.
  - "a" - Append - Opens a file for appending.
  - "w" - Write - Opens a file for writing.
  - "x" - Create - Creates the specified file.

Open a File

- To open a file for reading it is enough to specify the name of the file
- Create a new file named `data.txt`
- Add the following code to `main.py`

```
f = open("data.txt")
```

- The `open()` function returns a file object, which has a `read()` method for reading the content of the file

```
f = open("data.txt")
print(f.read())
```

  - Return the first 5 characters of the file

```
f = open("data.txt")
print(f.read(5))
```

Read Lines

- By looping through the lines of the file, you can read the whole file, line by line

```
file_data = open("data.txt")
for line in file_data:
  print(line)
```

Close Files

- It is a good practice to always close the file when you are done with it.
- In some cases, due to buffering, changes made to a file may not show until you close the file.

```
f = open("data.txt")
print(f.readline())
f.close()
```

Write to an existing file

- To write to an existing file, you must add a parameter to the `open()` function:
  - "a" - Append - will append to the end of the file
  - "w" - Write - will overwrite any existing content

Append data

- Add a new line to `data.txt`

```
f = open("data.txt", "a")
f.write("\nNow the file has more content!")
f.close()

# Open and read the file after the appending
f = open("data.txt", "r")
print(f.read())
```

Overwrite data

- Open the file `data.txt` and overwrite the content.

```
f = open("data.txt", "w")
f.write("I have deleted the content!")
f.close()

# Open and read the file after the appending:
f = open("data.txt", "r")
print(f.read())
```

Create a new file

- To create a new file in Python, use the `open()` method, with one of the following parameters:
  - "x" - Create - will create a file, returns an error if the file exists
  - "a" - Append - will create a file if the specified file does not exist
  - "w" - Write - will create a file if the specified file does not exist
- Create a file called "data.txt"

```
f = open("data.txt", "x")
```

- Create a new file if it does not exist

```
f = open("data.txt", "w")
```

Delete a file or folder

- To delete a file, you must import the OS module, and call `os.remove()` function.

```
import os
os.remove("data.txt")
```

- To delete an entire folder, call the `os.rmdir()` function.

```
import os
os.rmdir("test_folder")
```

  - You can only remove empty folders.

Test-Driven Development Demo

greetings.py

```
def get_greetings():
    return 'Hello World!'
```

main.py

```
import unittest
import greetings

class GreetingsTests(unittest.TestCase):

    def test_get_greetings(self):
      self.assertEqual(greetings.get_greetings(), 'Hello World!')

    def test_something(self):
      self.assertTrue(2<1)

if __name__ == '__main__':
    unittest.main()

# In jupyter notebook
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```