Python refresher - notes

Running Python

Interactive shell
- Invoke the Python interactive shell by typing `**python**` on the command line
- To open a file in the interactive shell Add it as a command line argument `**python myfile.py**`

Interactive notebooks
- **Jupyter notebooks** (Jupyter.org)
  - Think of this as an interactive shell that runs via a web client
  - *"The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text"* **(Jupyter.org)**
- **Anaconda**
  - Is a Python distribution which comes with a number of different tools (including a Jupiter notebook)

Beginner friendly Python

Note: This is a basic module, but there were a few things that I'd either forgotten or didn't know
- **'Pass'** statements can be used where the program expects a statement but none is required (see https://docs.python.org/3/tutorial/controlflow.html#pass-statements)

Data types - check with **type()**
- **Numerics** (integers, floats etc.)
  - **Note:** you can use _ within numeric literals to improve readability (See: https://www.python.org/dev/peps/pep-0515/)
- **Booleans**

Remember: Tuples and Strings are immutable, sets are not ordered

- **Sequences** (List, Tuple, Set, String)
  - There are several ways to create **multi-line strings** in Python, including docstrings (another way looks like a tuple without the commas)
  - **Negative indexing** - i.e. blah[-1]
  - **my_list.pop()** removes the last item (updating the list) and returns it).
  - **my_list.remove("blah")** will remove the given item from the list
- **Dict**
  - Add new properties with bracket syntax, i.e. **person['name'] = 'Boing'**
  - **Delete properties** with the del keyword, i.e. **del person['name']**
  - Get the keys with **keys()** and the values with **values()**
  - Use **get()** to retrieve an item by key.
    - This returns none when the key isn't found. Optionally takes a second parameter that can be the return value where a key isn't found.
  - Get the items with - you guessed it - **items()**
- **Tuple**
  - Much like lists but **immutable**
  - **dogs = ('Fido', 'Butch', 'Max')**
- **Sets**
  - **Mutable** sequences of **unique** items
    - Has a lot of useful methods
  - Sets **do not preserve their order**
  - **colours = {'blue', 'green', 'pink'}**
- **None**
  - None is a variable of the NoneType. It just means that the variable **contains nothing**
- **Files**
  - **It's easy to create, read and update files using Python**

Mutable vs Immutable
- Immutablity can improve performance and improve data integrity

Printing
- Prior to 3.6 people used **"".format(),**
  - but there are now **fstrings** - **f"Hello {name}"**

Intermediate Python

Comparison operators
- **is** keyword (for **identity**)

Loops
- **For**
  - Loops over an iterable
  - **Unpacking**

```
((30, 'Kalob'), (5, 'Zephyr'), (3, 'Henry'))

for age, name in people:
    print(f"{name} is {age} years old")

Kalob is 30 years old
Zephyr is 5 years old
Henry is 3 years old
```

Unpacking a tuple

  - **Unpacking a dictionary**

```
guitar = {
    "model": "Telecaster",
    "brand": "Fender"
}

for key, value in guitar.items():
    print(f"{key}:\t {value}")

model:   Telecaster
brand:   Fender
```

You use the .items() to return a list of tuples from the dictionary

- **While**
- **Break and continue**

```
while True:
    name = input("Guess my name: ")
    if name.lower() == "henry":
        print("YOU WIN")
        break

    print("That was the wrong name... try again!")
```

Continue skips the current iteration, Break jumps out of the loop

Type casting
- **list() int() etc...**

Helpful operators
- The **'in'** keyword
  - Works for iterables (strings, lists, etc.)
- **The enumerate function**

Enumerate

```
languages = ['python', 'javascript', 'c++', 'swift']

for index, lang in enumerate(languages):
    print(index, lang)

0 python
1 javascript
2 c++
3 swift
```

Enumerate provides an easy way to **get the current index of an item**

- **The Zip function**

Zipping

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]

for letter, number in zip(list1, list2):
    print(letter, " => ", number)

a  =>  1
b  =>  2
c  =>  3
```

When passed multiple *iterables* it returns a zip object - which is an *iterator* of tuples.

- **import**

Comprehensions
- **List comprehensions**
  - A way to make a for loop on one line. Always return a list

```
mylist2 = [letter for letter in course]
```

Basic

```
vowels = [letter for letter in course if letter.lower() in 'aeiouy']
```

With condition

```
c = [-40, -20, 0, 10, 15, 25, 35, 50]
f = [(temp*1.8) + 32 for temp in c]
print(f)
```

Performing calculations

```
keys = ['brand', 'model', 'year']
values = ['Gibson', 'Hummingbird', 2018]

guitars = {key:value for key, value in zip(keys, values) if key != 'year'}

print(type(guitars))
print(guitars)

<class 'dict'>
{'brand': 'Gibson', 'model': 'Hummingbird'}
```

Dictionary comprehensions

- See more here: https://www.python.org/dev/peps/pep-0274/

```
sum([x*x for x in range(10)])
```

There are also **generator comprehensions**

Functions
- def keyword
- Default arguments

Args and Kwags
- **How they differ**

Notice the use of '*' before args

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args('Computer', 'Coffee', 'Cup', 'Monitor', 'Lamp')

Computer
Coffee
Cup
Monitor
Lamp
```

Args are denoted with the * and **come through as an iterable (a tuple)**

Kwargs are denoted with ** and **come through as a dictionary**

- **General notes**
  - The names are arbitrary - you don't have to use the words args and kwags in your function definitions. What matters is the presence of * and ** before the names
  - **The order matters** *"In a function call, arguments must appear in this order: any positional arguments (value); followed by a combination of any keyword arguments (name=value) and the *iterable form; followed by the **dict form." (From Learning Python, Lutz)*

Comments
- **Docstrings - denoted with triple quotes**
  - **Allow for multi-line comments**

Map
- Loop through an iterable and provide a function for each item

```
for new_num in map(new_func, nums):
    print(new_num)
```

Has a different syntax to what you'd see in other languages

- This is also different in another important way: map() returns a map object - not a list. To use it you either need to iterate over it in a for loop or, alternatively, to cast it to a list.
  - Behind the scenes a map object will use a generator to get the values when cast or used within a for loop

Filter
- **Filter items in an iterables using a function**

```
for num in filter(filter_even_numbers, nums):
    print(num)
```

Within a for

```
evens = list(filter(filter_even_numbers, nums))
```

Cast to a list()

Lambda expressions

```
add = lambda num1, num2: num1+num2
```

Creating a one-time use function (similar to anonymous or arrow functions in JavaScript) which can be assigned to a variable

- These aren't used every day in Python. But you do use them

Scope

```
name = "Kalob" # Global
def greeting():
    name = "Zephyr" # Enclosing

    def say_hello():
        # Anything here would be local
        print("Name is ", name)

    say_hello()
```

Python uses the **LEGB** rule (this is a kind of lookup procedure, which starts with Local and ends with Built-in)

- Local (or function) scope
- Enclosing (or non-local) scope
  - An Enclosing scope only exists for nested functions. If there is no nested function, there is no enclosing scope.
- Global (or module) scope
- Built-in scope

Generators
- A generator is a Python sequence creation object. With it, you can iterate through potentially huge sequences without creating and storing the entire sequence in memory at once. Generators are often the source of data for iterators (from Introducing Python, 2nd Ed)
  - Note: once the sequence that has been created by a generator has been exhausted, it cannot be iterated over again
- **range()**, for example, is a generator
- **next()**
  - Gets the next value or throws an exception if the sequence is exhausted
- **iter()**
  - **The iter() function returns an iterator object**
  - **Iterable vs iterator**
    - An iterable is something that can be looped over
    - An iterator is an object that makes something loop able

Advanced Python

OOP

```
name = "Python"

print(name)
print(type(name))

Python
<class 'str'>
```

Most things in Python are objects

- **Classes**
  - Tend to use Pascal case (whereas snake casing is normal in Python elsewhere)
  - **__init__** is the constructor
    - Note: any time you see a **'dunder'** you know it's something Python provides
  - **self** refers to the instance
  - **Attributes**
  - **Methods**
    - You must use 'self' as the first argument
  - **Inheritance**

```
class Animal:

    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def speak(self):
        print("What does the animal say?")

class Cat(Animal):

    def speak(self):
        print("Meooowwww")
```

Achieved by **adding parens to the class definition and providing the class to be extended there**

  - **Interfaces**
    - An interface is a **collection of operations that do not have have an implementation.**
      - The most common use for an interface is to be used as an interface between applications, libraries or components
    - Some languages have a discrete keyword for creating interfaces. Python does not.
    - In Python, an interface is a class for which the methods either raise an exception or assert False (which in turn raises an exception)
    - Abstract Class Vs. Interface

Note: So, how do Interfaces different from Abstract Classes

  - **Super function**
    - **The super function allows you to call methods of a parent class**
    - **There is divided opinion about the use of the super() function.** See Chapter 32 of Lutz's Learning Python for a description (where he suggests referencing the given class methods directly, rather than via super(). His opinion is that *"super has substantial downsides in typical code. Most users are better served by the explicit name call scheme... despite its advocate's best intentions, super is not widely recognized as "best practice" in Python today, for completely valid reasons"*
  - **Dunder methods** (aka 'double underscore methods', sometimes called 'magic methods')
    - There are many of these. The course covers three.
      - **__init__()**
        - **Constructor**
      - **__str__()**
        - Use this to set the string representation of the given object (for example if you were to pass an object to print())
      - **__len__()**
        - You can use this to set what is considered the 'length' - i.e. what would be shown if you pass an object to len()
      - **__dict__()**
        - Returns a dict representation of your object

Packages
- **A package is a collection of modules. They usually exist in their own folder.**
  - **__init__.py**
    - The presence of this **file** tells Python that a given directory is a package. It can be an empty file.
    - **Notes on __init__.py**
      - **Until Python 3.3 these are required for a package import to work**
      - The __init__.py file **can contain code. If so it will be run upon import.** In this way the __init__.py serves as a hook for package initialisation-time actions (such as creating required data files or opening database connections)
- **Third party packages**
  - A strength of Python is the number of third-party packages
  - Typically hosted on PyPI (the Python Package Index)
  - **We use pip to install packages**
    - To see which version of pip you're using pass the -V flag to pip (i.e. pip -V)
    - Note: this will look for the version of pip associated with your version of Python. If you're one of those people who need to use python3 in the command line (as I am) then you would instead use pip3 -V
    - Use 'pip show [package]' to see version
    - Use 'pip freeze' to get a snapshot of all the snapshot of all your installed packages

A package is typically made up of many modules

Modules
- These are essentially .py files.

__name__ and __main__
- In every Python script there is a variable called __name__
  - If __name__ == "__main__" we know that it is a program running the file (i.e. Python running it directly). If __name__ is not "__main__" we know that this is an import

```
from first import scrape_a_page

def scrape_a_page(url):
    #
    # magic logic in here
    #
    return "something"


if __name__ == "__main__":
    # this is a program calling the file
    scrape_a_page()
```

What is the relevance of this? Well, what you'll often see is that where __name__ == "__main__" that a function will be called. This **allows you to use a given file as both an import and a regular program file**

Errors and Exceptions
- In Python we have **try and except** (as opposed to the 'catch' you see in other languages)
- We use the except block to gracefully handle errors and exceptions (rather than execution just stopping)
- **Keywords**
  - **Try**
  - **Except** (will run if there's an exception)
  - **Else** (will run if there are no exceptions)
  - **Finally** (will always run)

```
num1 = input("Enter the first number: ")
num2 = input("Enter the second number: ")

try:
    num1 = float(num1)
    num2 = float(num2)
    div = num1 / num2
    print(div)
except ValueError:
    print("There was a value error WOOPS")
except ZeroDivisionError:
    print("Cannot divide by zero")
except NameError as e:
    print("There was an undefined variable somewhere.", e)
except Exception as e:
    print(type(e))
    print(e)
else:
    print(f"Huzzah success! We divided {num1} and {num2}")
finally:
    print("This will always run no matter what")
```

Catching Exceptions

Unit testing
- **PyUnit** (called unittest in the library manual - see https://docs.python.org/3/library/unittest.html), provides an object-oriented class framework for specifying and customizing test cases and expected results. It mimics the JUnit framework for Java. This is a sophisticated class-based unit testing system; see the Python library manual for details.

```
def dog():

    def speak():
        return "Woof woof!"

    total = speak()

    return speak

woof = dog()

woof

<function __main__.dog.<locals>.speak()>

woof()

'Woof woof!'
```

Nested functions

- The same concept as closures in JavaScript
- Useful in Python for organising the logic within a function and for creating decorators

Decorators
- A decorator is a callable that takes another callable as an argument. It may augment or replace the function passed as an argument.
  - Essentially - what we're doing here is **providing a wrapper for the original function**
- Decorators allow us to keep our code **DRY**

Generators
- ***"A generator is a Python sequence creation object. With it, you can iterate through potentially huge sequences without creating and storing the entire sequence in memory at once. Generators are often the source of data for iterators...."*** From Introducing Python, 2nd Ed
- ***"Every time you iterate through a generator, it keeps track of where it was the last time it was called and returns the next value. This is different from a normal function, which has no memory of previous calls and always starts at its first line with the same state."*** From Introducing Python, 2nd Ed

```
def my_range(first=0, last=10, step=1):
    number = first
    while number < last:
        yield number
        number += step
```

- **Simple generator**
- The sequences created by a generator are good for one pass. After they've been exhausted there's nothing to iterate over
- **next()**
  - Will get the next value from the sequence resulting from a generator. Once the sequence is exhausted an exception will be thrown.
- **Generator comprehensions**
  - Looks like other comprehensions but is surrounded by parentheses

Linting
- **Isort** - will sort your imports correctly
- **flake8** - will check files for unused imports etc. You will then go in an make the changes yourself.
- **black** - is an uncompromising code formatter. It is opinionated. It will manage line length, manage quotas vs apostrophes etc.

Virtual environments
- Allow you to have different versions of Python and packages per project. Everything you do in Python should ideally be done within a virtual environment
- Two popular ways to create them
  - **venv** - built into Python
  - **pipenv** - newer, arguably easier way to create virtual environments. Requires installation.
    - Differences:
      - Allows you to specify a different version of Python
      - Once inside a pipenv environment you install dependencies with `pipenv install` rather than `pip install`
      - You use `pipenv shell` to enter the environment (rather than source ... activate)
      - You use Ctrl+D to leave the environment (rather than using 'deactivate')

Requirements files
- Usually a **requirements.txt** file
- Requirements can be installed with **`pip install -r requirements.txt` (**do this within a VE to ensure they are not installed to your local machine)
- When you have installed new dependencies you can save these to your requirements.txt file with `pip freeze > requirements.txt`

Interactive python
- ipython is a more convenient version of the interactive shell

Python environments
- pyenv allows you to have multiple version of Python

Simple web server
- Good for serving static files
  - **python3 -m http.server [port]**