Introduction to Python

Python 3

- Powerful programming language but easy to write and understand
- Became de facto a standard in Data Science/Machine Learning
  - extensive number of libraries for DS that are available
  - notebooks
- Script language, no need to compile code
  - Same code works on different OSs and machines
  - But requires an interpreter to be installed

Basics - Variables

- Variables - a place to hold some value
  - Boolean **`True`**, **`False`**
  - Integer …., `-2`, `-1`, `0`, `1`, `2`, ….
  - Float `0.5772`, `1.618`, `3.1415`, etc.
  - String `"some text"`, `"another text"`, etc.
- **`None`** - special value indicating lack of any value
- Python is loosely typed language
  - Type of variable does not have to be specified
  - Variable may be used to store different types
  - But you still may want to cast to different type, e.g. `integer = int("12")`

Basics - Operators

- Operators enable operations on or using variables
- Arithmetic operators
  - **+**, **-**, *****, **/** add, subtract, multiply, divide
  - **%** modulus returns reminder of division e.g. `8%3` => `2` (because 8 = 2*3 + **2**)
  - ****** exponent, e.g. `2**3` => `8` (as $2^3 = 8$)
  - **//** floor division - returns closest integer smaller than division result, e.g, `9//2` => **4**
- Assign operators
  - **=** assign new value to a variable
  - **+=**, **-=**, …. So called in-place operators, they enable updating current value of a variable

```
some_text = "some text"
a = -2
a += 2
b = 2 * 10 - 1 / 2 + a
```

Basics - Operators

- Comparison (results in **`True`** or **`False`**)
  - **`<, <=, >, >=`** less than, less or equal, etc.
  - **`==`** equal
  - **`!=`** or **`<>`** not equal
- Logical (results in **`True`** or **`False`**)
  - **`or`**, **`and`**, **`not`**, **`!`**
- There are more operators (like bitwise operators)
  - List of all python 3 operators: https://docs.python.org/3/library/operator.html#mapping-operators-to-functions
- Advanced: In python operators can be overloaded
  - It means that a programmer can define what operations they will do
  - for string **+** is overloaded to concatenate two strings
  - in TensorFlow/numpy **`+`** is overloaded to add matrices, **`@`** is overloaded to multiply them

Basic - printing

- It is useful to print messages to monitor progress/outcome of a script
    - To print a message in python use `print()` function
    - Put your message/variable/value between `()` brackets
    - E.g. `print("Hello World!")`
- Make use of formatted strings introduced in python 3
    - If you would like to print variable's value, e.g. `name = "Marcin"`
    - Use `f` in front of a string
    - Then embed python code inside string using `{}` brackets
    - E.g. `print(f"Hello {name}!")` will print `Hello Marcin!`
    - `print(f"{2 + 2}") => 4`
    - `print(f"{5 + 1} = {2 + 2}? {6 == 4}" ) => 5 == 4? False`

Notebooks

- Notebooks are great way to code in Python
  - Composed of code and text cells
  - May provide rich output for a cell (interactive tables and graphs)
  - Very useful for exploratory data analysis or simple tasks
  - Last line of code block will be printed
- They have their dark side too
  - They are not suitable for developing more complex applications
  - IDEs (Integrated Development Environments) provide better code auto-completion
  - They allow you to execute code in any order...
- It is important to understand how state works
  - Execute cells in order
  - Avoid reassigning variables (unless you know what you are doing)

Google Colab

- Google colab is a service for editing, storing, and running notebook
    - You can also collaborate with friends on the same notebook
- The code is execute on google cloud
    - It doesn’t have access to your local computer/drive!
- There are some tricks that you can do with them that we will explore throughout this course
    - Creating forms (for configuring input parameters)
    - Uploading file to cloud
    - Installing new libraries
    - Using GPUs
- Learn more here: https://colab.research.google.com/notebooks/welcome.ipynb

Colab, 1st exercise

In this exercise we will get familiar with colab, define and print simple variables, and use forms to parameterize input.

To start please create account on colab you can use your personal gmail/ualberta account) https://colab.research.google.com/

Then load provided `exercise-1.ipynb` file and follow the instructions

Flow control - conditional execution

- Execute different code depending on a condition
    - **`if`**, **`elif`**, **`else`** chain
- Code blocks
    - In python indentation (typically 4 spaces) defines code blocks
    - **`pass`** - special keyword to use if we want to leave code block as empty
- One line conditionals
    - `larger_value = a if a > b else b`
    - `if a > b: print("a is larger")`

```
if a > 2:
    print("a is larger than 2")
elif a <= 0:
    pass
else:
    print("a is between 0 and 2")
```

Flow control - loops

- Execute code multiple times
  - **`for`** `i` **`in`** *`sequence`*`:` executes block of code for every element in sequence
  - **`while`** *`condition`*`:` executes block of code until condition is true
- Loop control
  - **`break`** breaks the loop
  - **`continue`** skips the remaining code in block and executes next iteration f the loop
- Range
  - **`range([start=0,] end[, step=1])`** generates a sequence of numbers (from start to end (exclusive) with a given step)
  - **`range(6) => 0, 1, 2, 3, 4, 5`**

Colab, 2nd exercise

In this exercise we will practice if-elif-else and loops

Please load provided `exercise-2.ipynb` file and follow the instructions

Collections - Lists & Tuples

- Lists and tuples are used to store multiple values
    - List can be modified, tuples once created cannot change
    - Values can be of any type
    - Both are indexed (values can be accessed using **`[index]`**)
        - Indices start at 0
- Initializing
    - `a_list = [1, 2, 3]`
    - `a_tuple = (1, 2, 3)`
    - `a_list = list((1, 2, 3))`
- Modifying lists
    - `a_list.append(value)` adds a value to the end of list
    - `a_list.pop([pos=-1])` removes a value from a specific position
    - `a_list.remove(value)` removes first occurence of a specific value
    - `a_list.extend(b_list)` extends list with elements from `b_list`

Collections - Lists & Tuples

- To access a value in a list or a tuple use `[]` brackets
  - `a_list[0]` returns first value in a list/tuple
  - `a_list[-1]` returns the last value in a list/tuple
- To access multiple values at once use slices `[start:end:step]`
- For a list `[0, 1, 2, 3, 4, 5, 6, 7, 8]`:
  - `[:]    => [0, 1, 2, 3, 4, 5, 6, 7, 8]`
  - `[1:3]  => [1, 2]`
  - `[-2:]  => [7, 8]`
  - `[:5:2] => [0, 2, 4]`
  - `[::2]  => [0, 2, 4, 6, 8]`

Collections - Sets

- Set stores only unique values
  - Values can be added or removed through **`add(`**`value`**`)`** or **`remove(`**`value`**`)`** methods
  - Values are not stored in order they were added
- Additionally common set operations can be performed
  - `set1 = set([1, 2, 3, 4])`
  - `set2 = set([2, 4, 6, 8])`
  - `set1.union(set2)        => set([ 1, 2, 3, 4, 6, 8])`
  - `set1.intersection(set2)=> set([ 2, 4])`
  - `set1.difference(set2)  => set([ 1, 3])`
  - `set2.difference(set1)  => set([ 6, 8])`
- There is no concept of indexing and hence accessing elements using **`[]`** does not work with sets

Collections - Dictionaries

- Dictionaries store key value map
  - Values may be anything, e.g.: string, integer, list, tuple
  - Whereas keys must be immutable and hashable, e.g.: string, integer, tuple
  - There will be only one entry for a given key (so there is a **set** of keys)
- Dictionaries are defined using **`{}`** brackets or **`dict()`** function
  - `a_dictionary = {}`
  - `a_dictionary = {1:1, "a": 2, "c": [1, 2, 3]}`
  - `a_dictionary = dict([("a", 1), ("b", 2)])`
- `[]` brackets can be used to get a value for or assign value to a given key
  - `a_dictionary["a"]`
  - `a_dictionary["a"] = 2`

Collections - Iterating through values

- **`for`** `value` **`in`** *`collection`*`:`
  - list or tuple - goes through each element in the list from first to last
  - set - goes through each element in set, but in undetermined order
  - dict - goes through all keys (in the order they were added to dictionary)
  - For dict use **`.items()`** or **`.values()`** to access tuples (key-value pairs) or just values
- Use **`zip()`** function to iterate over two or more collections at once
  - **`for`** `a, b` **`in zip`**`(a_list, b_list):`
  - Note: In example above `` `a, b` `` actually denotes tuple
- To check if a given value is in a collection use **`in`** operator
  - For dictionaries this will only check if a key is defined in a dictionary
  - For large lists it may be costly operation if repeated multiple times

Advanced: List Comprehensions

- A shorthand for creating lists
- `[ expression for value in collection ]`
  - `a_list = [ i**2 for i in range(5) ] => [0, 1, 4, 9, 16]`
- This can also be used to create dictionaries
  - `d = dict((i, i**2) for i in range(5)) => {0:0, 1:1, 2:4, 3:9, 4:16}`
  - **`dict`** function creates a dictionary from a list of tuples
  - Note: in this example there is no `[]`, this prevent creation of unnecessary list
- … or tuples or sets using **`tuple()`** or **`set()`** functions

Colab, 3rd exercise

In this exercise we will practice working with lists.

Please load provided `exercise-3.ipynb` file and follow the instructions

Structure - Functions

- There is often a need to run the same code multiple times
  - Finding maximum/minimum in list
  - Performing some complex calculations
  - Handling adding element to a list…
- Functions are great way of reusing code
  - There is a few built in functions in python like `range()`, `print()`, `max()`, `min()`, etc.
  - There are also functions in collections `a_list.append()`, `a_list.extend()`, etc.
  - Functions that are executed on objects (using . in front) are called methods, more about it soon
- Python enables to define functions quite easily

Structure - Functions

```
def function_name(parameter1, parameter2, ...):
    # function body
    return
```

- Functions are defined using **`def`** keyword followed by function name
  - Make name informative
  - No spaces allowed
- Function name is followed by comma separated list of parameters
  - This is function input, data needed to perform operations defined inside function
  - There may also be no parameters
- Function body is defined using indentation
- Function may or may not have a return statement

Structure - Functions

- Use a function’s name and pass arguments to run the function’s code
    - `a_function(1)`, `b_function()` , etc.
- Function will return values as defined in return statement
    - There may be no return statement in which case function will return None
- Some parameters may have default values
    - Parameters without default value should be listed first
    - To assign default value use = ,e.g., `def a_function(param1, param2=1):`
    - When calling function the parameters with default value may be omitted
    - `a_function(1)` is equivalent to `a_function(1, 1)`
- You can use parameter names when calling function to add some clarity
    - `a_function(param2=3, param1=1)` is equivalent to `a_function(1, 3)`

Advanced: Structure - Functions

- Recursion: A function can be from within itself

```
def factorial(n):
    if n < 1:
        return 1
    else:
        return n * factorial(n - 1)
```

- Lambda function
  - A way to define an in-place function to avoid defining a function that will be used in one place
  - e.g. `a_list_of_tuples.sort(key= lambda tuple: tuple[1])`
- Two functions cannot have the same name
  - Unless they are in different namespaces (classes, modules, etc.)
  - But functions can be defined with variable number of parameters

Structure - Modules

- Code in python is organized in modules
  - Typically module should be focused on a specific area/function
- Libraries are composed of one or many modules
  - E.g. pandas, scikit, tensorflow, etc.
- Module defines functions and classes that can be used in other modules
- Modules are imported using **`import`** and **`from`** keyword
  - `import math`
    - Then in code: `math.sqrt(2)`
  - `import pandas as pd`
    - Then in code: `pd.read_csv("dictionary.csv")`
  - `from sklearn.tree import DecisionTreeClassifier`
    - Then in code: `tree = DecisionTreeClassifier(max_depth= 3)`

Advanced: Structure - Classes

- Classes are useful to keep a state (of some variables) and enable certain functionality (behavior)
  - Class is a definition and actual instance/variable is called an object
- List is a good example of a class
  - State: elements that are in the list
  - Behaviour: all the operations that were discussed
- Class can inherit and extend behaviour from a parent class
  - Useful when you want to create a few classes that share some state/behaviour - they may have a single parent

Advanced: Generators and iterables

- `for` loops actually go over iterable values (and collections are iterable)
- A class is iterable if it implements `__iter__(self)` and `next(self)` methods
- Another way of providing elements to iterate is to define a generator:
    - `yield` keyword yields a value and then when code that called generator is done with processing that value it returns to the same place in code inside generator

```
def positive_integers(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for i in positive_integers(5):
    ...
```

Colab, 4th exercise

In this assignment we will refactor our code from the previous exercises and add some functionality.

Please load provided `exercise-4.ipynb` file and follow the instructions

Strings

- Strings are one of basic variable types and they allow working with text
- Text value has to be surrounded with `"` or `'`
  - They are equivalent, choose one and keep using it
- There are a few special characters combinations
  - `\n` new line character
  - `\t` tab
  - `\r` carriage return (on windows new line is \r\n, on old MacOS \r)
  - `\\` to print \
  - `\"` `\'` for " and '
- Strings cannot be simply broken into multiple lines
  - To break long strings into multiple lines use `\` after string definition, e.g., `""\`
  - Or use `"""` around multiline text, however, this will preserve line breaks in resulting string

String - functions

- Python has a variety of useful functions for strings:
  - `.strip()` removes leading and trailing spaces
  - `.split(sep)` splits a string, on each occurence of `sep`, into list of strings
  - `.find(substr[, start=0, end=-1])` returns index of substr in a given string (or -1)
  - `.replace(old, new[, max])` replaces up to `max`/all occurrences of `old` with `new`
  - `.lower()`, `.upper()` converts the character to all lower or all upper
  - and more https://docs.python.org/3/library/stdtypes.html#string-methods
- Slices can also be used on strings
  - `"abc"[0] => "a"`
  - `"abc"[0:2] => "ab"`
- … and global functions and keywords
  - `"ab" in "abc" => True, "a" not in "abc" => False`
  - `len("abc") => 3`

File IO

- Working with files in Python is easy

```
with open("file.txt", "r") as input_file:
    for line in input_file:
        ....
```

- When writing to file use `"w"` mode
  - r - read
  - w - write
- Colab is working in cloud so files has to be uploaded first
  - The code to upload files to the colab is provided in the 5th exercise

Colab, 5th exercise

Load data from file containing results from experiment, extract relevant information from text, and calculate score for each setup and rank them according to score.

Reuse code from previous exercises.

Code readability

- Comments help to understand code
  - They shouldn't be used extensively
  - People often forget to update them when they update code
  - Use docstrings to provide details about function purpose input and output
- Text cells in notebooks are a good place for providing comments/narrative
- Write self-documenting code
  - Use good, descriptive, names for variables, functions, etc.
  - Keep code blocks/functions small and responsible for one function
- Follow code style convention
  - Popular code style guide for python: pep8 https://www.python.org/dev/peps/pep-0008/
- Advanced: Use flake8 to validate that code style adhere to pep8 style guide
- Advanced: Use type hints https://docs.python.org/3/library/typing.html

Where to go from here?

Classes

Regex

Handling exceptions

Unit tests

Stackoverflow

Virtual environments (venv)

Source versioning