python_part3

November 11, 2020

1 Introduction to Python - Part 1

Python is called an interpreted language but is compiled to bytecode. The .py source code is first compiled to byte code as .pyc. This byte code can be interpreted (official CPython), or JIT compiled (PyPy).

1.1 Jupyter notebook

1.1.1 Useful to write both code and text in the same place

Double click on a cell, choose in the top menu either "code" or "Markdown" for text. After writing something in the cell, just press `shift` + `Enter` to view the result (or click on the "Run" button)

In addition of headings (#, ##, ###, ...) with markdown you can write bullet lists (*)

- Use #, ##, ###, etc. on the first line for the size of your headings
- Use single star (*) for each entry of your bullet list

Headings are obviously optional. In addition of bullet lists, you can create numbered lists using the format "number" "fullpoint(.)" "space":

1. Markdown allows you to write simple text
2. But also LaTeX formulas, by simply putting it between dollar signs: $\sum_{n=1}^{\infty} \frac{1}{2^n} = 1$
3. You can escape special markdown characters with a backslash (\): *
4. You can even quote commands by using apostrophes (‘), as in: use `jupyter notebook` to launch Jupyter notebook from the terminal (after installing python and Jupyter)
5. You can put words in *italics* by writing them in between stars (*) or in **bold** using two stars (**)

[4]:
```
# Now let's do some python code. Comments can be made with a hash sign (#).
2 + 5
```

[4]: 7

While the result of executing a cell (`shift` + `Enter`) in *Markdown* is **formatting that cell**, for *code* (like `python` code) the result is **running that code and displaying the output (if any)**

The jupyter notebook can be in either of two modes:

- **edit mode** when you are editing a cell (i.e. typing Markdown or code): **the edited cell is green**
- **command mode** when you can can execute commands from the keyboard (e.g. adding or deleting a cell): **the cell marker becomes blue**

To go from edit mode to command mode, press the `Esc` key. To go from command mode to edit mode, press the `Enter` key. This is useful to avoid using the mouse and thus saving time.

Press the keyboard button on the menu at the top to see all keyboard shortcuts.

Exercise

1. Create two new cells clicking on `+` in the top left corner, once for each cell
2. Make the first cell a Markdown cell, by clicking on it and selecting `Markdown` in the menu (top center)
3. Edit it by double-clicking it, then write a bullet list of items
4. Make the second cell a code cell, by clicking on it and selecting `code` in the menu (top center)
5. Edit it by double-clicking it, then write an arithmetic operation
6. Make sure to execute both cells (if not already done) by selecting it and pressing the "Run" button (top left)

Exercise (part 1) Do the same as above but without using the mouse except for the initial step `0`.

0. Click on the last cell you created and make sure the cell marker is blue (otherwise press the `Esc` key)
1. Create two new cells by pressing twice on `b` on the keyboard
2. Navigate to the first cell you created using the up / down arrows on your keyboard
3. While in command mode (make sure the cell marker is blue otherwise press the `Esc` key), press the `m` key to make the cell a Markdown cell
4. Switch to edit mode by pressing the `Enter` key, then write a numbered list of items
5. Execute the cell by pressing `Shift` + `Enter`: since the cell is a markdown cell this will format the cell

Exercise (part 2)

6. Executing the previous cell will bring you to the next cell. While in command mode (make sure the cell marker is blue otherwise press the `Esc` key), press the `y` key to make the cell a code cell
7. Switch to edit mode by pressing the `Enter` key, then write an arithmetic operation
8. Execute the cell by pressing `Shift` + `Enter`: since the cell is a code cell this run the code inside the cell

```
[ ]:
```

1.2 Python variables and datatypes

Variables allow for generalization and reusability of programming instructions. A variable is a reference to an object. The object can be:

- A sequence of characters, i.e. a "string":

  - `my_var = "Felix"`
  - `my_var = "My cat is called Felix"`
  - `my_var = "/home/felix/how_to_manipulate_humans.txt"`
- A number (integer, float)
  - `my_var = 3.14`
- A list of objects:
  - `my_vars = ["felix", "/home/felix/file.txt", 3.14]`

Variable names

- can only contain letters, digits, and underscore `_` (no spaces!)
- cannot start with a digit
- are case sensitive (`fruit`, `FRUIT` and `Fruit` are three different variables)

Let's look at some code. In python spaces can be used on each side of equal sign, and the content of the variable is simply obtained by indicating the variable name (no need for `$` like in bash).

[13]:
```
# a refers to an integer (int)
a = 2
a
```

[13]:
```
2
```

Functions (to be discussed later) are like commands in bash, but take arguments inside parentheses.

[8]:
```
# simple_pi refers to a float
simple_pi = 3.14
print(simple_pi)
```

```
3.14
```

Compare with the bash version:

[16]:
```
%%bash
pi=3.14
echo ${pi}
```

```
3.14
```

As in bash, strings (i.e. sequence of characters) are indicated inside quotes

[9]:
```
# favorite_fruit refers to a string (str)
favorite_fruit = "kiwi"
print(favorite_fruit)
```

```
kiwi
```

Python can define a boolean value. A boolean is a variable which can only have one of two values: `True` or `False`. For convenience `0` is considered `False` while anything different from `0` is considered `True`.

[10]:
```
# like_kiwi refers to a boolean
like_kiwi = True
print(like_kiwi)
```

```
True
```

You can check the type of a variable with `type`.
Note that in Python arguments to a function are separated by `,` and spaces can be used before (not recommended) or after (recommended) the comma.

[11]:
```
print(type(a), type(simple_pi), type(favorite_fruit), type(like_kiwi), sep=' ')
```

```
<class 'int'> <class 'float'> <class 'str'> <class 'bool'>
```

[5]:
```
print?
# Output: print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)␣
↪[can also use "Shift"+"Tab" for help]
```

Simple datatypes are **immutable** (you cannot change them)

[ ]:
```
# Setting a variable to a new immutable object (e.g. integer) will refer to a␣
↪new object entirely
a = 3
a = 4
```

[23]:
```
# Consider a variable 'a' referring to an immutable object (e.g. an int).
# Setting a variable 'b' to 'a' can also be thought as creating a new object,␣
↪with the same value but independent
a = 3
b = a
print('a is', a)
print('b is', b)
```

```
a is 3
b is 3
```

[25]:
```
# Since they are completely independent, changing one will not affect the other
a = 3
b = a
a = 4
print('a is', a)
print('b is', b)
```

```
a is 4
b is 3
```

[26]:
```
# Strings, similarly to list (cf next section), are 0-indexed sequences
atom_name = 'helium'
```

```
print('letter at index 0 is', atom_name[0])
print('letter at index 4 is', atom_name[4])
```

```
letter at index 0 is h
letter at index 4 is u
```

[27]:
```
# Do not forget that strings are immutable (you cannot change them)
atom_name[1] = 'i'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-81394bd6cbfd> in <module>
      1 # Do not forget that strings are immutable (you cannot change them)
----> 2 atom_name[1] = 'i'

TypeError: 'str' object does not support item assignment
```

[28]:
```
# However you can do many things with strings, for example you can concatenate␣
 ↪them
first_name = 'John'
last_name = 'Doe'
full_name = 'Mr' + ' ' + first_name + ' ' + last_name
print(full_name)
```

```
Mr John Doe
```

[29]:
```
# You can slice them, i.e. extract several sequential characters at a time
# <sub_string> = <main_string>[start:stop] where stop is the index AFTER the␣
 ↪last wanted character
red_fruit = 'strawberry'
sub_string = red_fruit[2:5]
print('sub_string is:', sub_string)
```

```
sub_string is: raw
```

[30]:
```
# You can ommit the starting index if you want to start from the beginning
# Similarly, you can ommit the stop index if you want to stop at the end
red_fruit = 'strawberry'
sub_string = red_fruit[5:]
print('sub_string is:', sub_string)
```

```
sub_string is: berry
```

1.3 Python data structures

Data structures prove often essential to write any kind of code, to facilitate reasoning / processing (hence shopping list and calories table)

1.3.1 Lists

[31]:
```
fav_fruits = ['kiwi', 'mango']
print('fav_fruits:', fav_fruits)
```

```
fav_fruits: ['kiwi', 'mango']
```

[32]:
```
fav_fruits_cals = [42, 60]
print('fav_fruits_cals:', fav_fruits_cals)
```

```
fav_fruits_cals: [42, 60]
```

[33]:
```
fav_fruits[1]
```

[33]:
```
'mango'
```

To use functions on objects such as lists, you can use either:

- **general functions** with `<function_name>(<object_name>)` such as `len(fav_fruits)` (to compute length with `len`), OR
- **specific object functions**, called **methods**, with `<object_name>.<method_name>` such as `fav_fruits.extend(['apple', 'pear'])` (to append a list of new elements to the original list with `extend`)

[16]:
```
len(fav_fruits)
```

[16]:
```
2
```

To see the list of all methods applying to an object, use general function `dir`. You can also use "TAB" after typing a dot (".") after your variable name.

[34]:
```
dir(fav_fruits)
```

[34]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
```

```
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

[ ]:
```
fav_fruits.
```

[35]:
```
fav_food = fav_fruits
fav_food.extend(['4cheese_pizza', 'banana_split'])
print('fav_food:', fav_food)
```

```
fav_food: ['kiwi', 'mango', '4cheese_pizza', 'banana_split']
```

[36]:
```
fav_food_cals = fav_fruits_cals
fav_food_cals.extend([1200, 900])
print('fav_food_cals:', fav_food_cals)
```

```
fav_food_cals: [42, 60, 1200, 900]
```

[37]:
```
total_fruit_diet_cals = sum(fav_fruits_cals)
print('total_fruit_diet_cals:', total_fruit_diet_cals)
```

```
total_fruit_diet_cals: 2202
```

1.4 WARNING: Lists are mutable

[38]:
```
print('fav_fruits:', fav_fruits)
print('fav_fruits_cals:', fav_fruits_cals)
```

```
fav_fruits: ['kiwi', 'mango', '4cheese_pizza', 'banana_split']
fav_fruits_cals: [42, 60, 1200, 900]
```

[39]:
```
fav_fruits = ['kiwi', 'mango']
fav_fruits_cals = [42, 60]
# Copy then extend fav_food and fav_food_cals
fav_food = fav_fruits.copy() # copy
fav_food.extend(['4cheese_pizza', 'banana_split'])
fav_food_cals = fav_fruits_cals.copy() # copy
fav_food_cals.extend([1200, 900])
print('fav_fruits: ', fav_fruits)
print('fav_fruits_cals ', fav_fruits_cals)
total_fruit_diet_cals = sum(fav_fruits_cals)
print('total_fruit_diet_cals:', total_fruit_diet_cals)
```

```
fav_fruits:  ['kiwi', 'mango']
fav_fruits_cals  [42, 60]
total_fruit_diet_cals: 102
```

[40]:
```
# Typically there are no problems such as above if you do not initialize a list␣
↪with another
fav_fruits = ['kiwi', 'mango']
fav_food = fav_fruits + ['4cheese_pizza', 'banana_split']
print('fav_fruits: ', fav_fruits)
print('fav_food: ', fav_food)
```

```
fav_fruits:  ['kiwi', 'mango']
fav_food:  ['kiwi', 'mango', '4cheese_pizza', 'banana_split']
```

[41]:
```
# Python overloads traditional operators in context you may not have think of
print(['we', 'will'] * 2)
```

```
['we', 'will', 'we', 'will']
```

[42]:
```
print(['we', 'will'] * 2 + ['rock', 'you'])
```

```
['we', 'will', 'we', 'will', 'rock', 'you']
```

1.5 Dictionaries

[26]:
```
food_cals = {
          'kiwi': 42,
          'mango': 60,
          '4cheese_pizza': 1200,
          'banana_split': 900
          }
```

[27]:
```
food_cals['mango']
```

[27]:
```
60
```

[28]:
```
food_cals.keys()
```

[28]:
```
dict_keys(['kiwi', 'mango', '4cheese_pizza', 'banana_split'])
```

[29]:
```
food_cals.values()
```

[29]:
```
dict_values([42, 60, 1200, 900])
```

[30]:
```
food_cals[1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-30-69ae36855740> in <module>
----> 1 food_cals[1]

KeyError: 1
```

1.6 WARNING: Dictionaries are mutable

1.7 Tuples: like lists but immutable

[1]:
```
my_experiment_params = (42, 'method3', 3.14, 'paramx', 'paramy')
office_lat_long = (46.2221685, 6.1482567)
```

[33]:
```
dir(office_lat_long)
```

[33]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
```

```
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'count',
 'index']
```

[2]:
```
# Unpack tuple (number of variables on the left should be equal to len(tuple))
office_lat, office_long = office_lat_long
```

Using the format function

[35]:
```
print('My office latitude is {} and its longitude is {}'.format(office_lat,␣
 ↪office_long))
```

```
My office latitude is 46.2221685 and its longitude is 6.1482567
```

[36]:
```
print('My office latitude is {lat} and its longitude is {long}'.
 ↪format(lat=office_lat, long=office_long))
```

```
My office latitude is 46.2221685 and its longitude is 6.1482567
```

Using so called "f-strings" (note the f in front of the string)

[3]:
```
print(f'My office latitude is {office_lat} and its longitude is {office_long}')
```

```
My office latitude is 46.2221685 and its longitude is 6.1482567
```

1.7.1 Exercise

1. Create a variable named `my_fruit` referring to the string 'pineapple'
2. Using indices, slice (i.e. extract) 'pine' from your variable
3. Use either `dir` or "TAB" (after writing '.' after your variable name) to display all methods applicable to your variable `my_fruit`
4. Use the list above to find a method to test if what is in your variable `my_fruit` ends with `apple`

1.7.2 Exercise

1. Create a dictionary `wish_list` representing a list of items you'd like with item names as keys and item prices as values
2. Using your dictionary `wish_list`, print the list of item names
3. Using your dictionary `wish_list`, print the total cost of your wish list (hint: use the function `sum`)

1.8 Control structures

Most programs need to do repeated actions, and perform different actions according to different conditions. That is why control structures such as "for loop" and "if else statements" are essential.

1.8.1 For loops

A for loop will iterate for each item in a sequence. A typical example is iterating over a list of integers.

[54]:
```
seq = [0, 1, 2, 3]
for i in seq:
    print(i)
```

```
0
1
2
3
```

In Python, a built-in function `range` is typically used to generate this sequence. Only indicating the integer `n` to stop at (exclusive) is enough, such as in `range(n)`. Both starting and stopping integers could be used, as in `range(k, n)`. And in the latter case, a `step` can even be set, as in `range(k, n, step)`.

[146]:
```
n = 3
for i in range(n): # same as: for i in range(0, n)
    print(i)
```

```
0
1
2
```

[147]:
```
n = 3
for i in range(1, n+1): # remember that stopping integer is exclusive
    print(i)
```

```
1
2
3
```

[149]:
```
n = 3
for i in range(0, n, 2): # the index increase by 2 at each iteration
    print(i)
```

```
0
2
```

The "for block" is indicated by an "indentation" (typically 4 spaces)

[156]:
```
n = 3
for i in range(n):
    print(i)
    print('launch!')
```

```
0
launch!
1
launch!
2
launch!
```

[157]:
```
n = 3
for i in range(n):
    print(i)
print('launch!')
```

```
0
1
2
launch!
```

The sequence does not have to be a sequence of integers, it can be any "iterable".

[4]:
```
fav_fruits = ['kiwi', 'mango', 'papaya']
for fruit in fav_fruits:
    print(fruit)
```

```
kiwi
mango
papaya
```

Python allows for concise and elegant syntax and this shows when extracting both the iterating index and the corresponding item.

[154]:
```
# Other programming language
n = len(fav_fruits)
for i in range(n):
    fruit = fav_fruits[i]
    print('Rank {}: {}'.format(i+1, fruit))
```

```
Rank 1: kiwi
Rank 2: mango
Rank 3: papaya
```

[5]:
```
# Python uses the built-in function "enumerate" to extract both index and item␣
↪at the same time
for i, fruit in enumerate(fav_fruits):
    print(f'Rank {i+1}: {fruit}')
```

```
Rank 1: kiwi
Rank 2: mango
Rank 3: papaya
```

This shows as well when iterating over dictionary items.

[158]:
```
food_cals = {
          'kiwi': 42,
          'mango': 60,
          '4cheese_pizza': 1200,
          'banana_split': 900
          }
```

[159]:
```
# Other programming language
all_food_names = food_cals.keys()
for food_name in all_food_names:
    n_cals = food_cals[food_name]
    print('Food item {} has {} calories'.format(food_name, n_cals))
```

```
Food item kiwi has 42 calories
Food item mango has 60 calories
Food item 4cheese_pizza has 1200 calories
Food item banana_split has 900 calories
```

[160]:
```
# Python extract directly the (key, val) pairs with the "items" method function
for food_name, n_cals in food_cals.items():
    print(f'Food item {food_name} has {n_cals} calories')
```

```
Food item kiwi has 42 calories
Food item mango has 60 calories
```

```
Food item 4cheese_pizza has 1200 calories
Food item banana_split has 900 calories
```

1.9 If - else statements

An "if - else" statement allows to execute commands only if a logical expression is True. The logical expression is called "Boolean expression" and is always either True or False. If False the block of commands is not executed.

[55]:
```
# Exam for which grade < 5 is fail, grade = 5 requires to resit exam, and grade
↪> 5 is a pass
grade = 7
if grade < 5:
    print("Grade too low")
```

An `else` statement is executed if the `if` statement is false

[165]:
```
grade = 7
if grade < 5:
    print("Grade too low")
else:
    print("Grade high enough")
```

```
Grade high enough
```

One or more `elif` statements can be added to add tests. Tests are always evaluated only once, in order.

[235]:
```
grade = 7
if grade < 5:
    print("Grade too low")
elif  grade == 5:
    print("Please resit exam")
elif grade < 8:
    print("Grade high enough")
elif grade <= 10:
    print("Excellent grade")
else:
    print("Your grade is greater than 10!")
```

```
Grade high enough
```

Boolean expressions can be combined with boolean operators: `and`, `or` or `not`. Below is an example with a dictionary having as values other dictionaries (values can be any objects).

[178]:
```
shopping_list = {
               'kiwi': {'cals': 42, 'price': 2},
               'mango': {'cals': 60, 'price': 5},
               '4cheese-pizza': {'cals': 1200, 'price': 18},
               'banana split': {'cals': 900, 'price': 9}
```

```
            }
```

[234]:
```
for food_name, food_info in shopping_list.items():
    if (not food_info['cals'] < 500) and (food_info['price'] < 15):
        print('Dinner tonight could be {food_name}')
```

```
Dinner tonight could be banana split
```

1.9.1 Exercise

1. Create a variable 'my_dict' referring to a dictionary (any content is fine)
2. Loop on all methods available for this dictionary, and only print the ones NOT starting with an underscore (_)
3. Create a variables `methods` listing all methods applying to your dictionary (hint: use the `dir` function)
4. Create a variable `proper_methods` set to an empty list (you will fill it next)
5. Create a for loop on all the items in the `methods` variable and append it to the list `proper_methods` if it does not start with an underscore

2 Introduction to Python - Part 2

2.1 Functions

A function allows you to group *a logical unit set of commands*, which has a specific aim, into **a single entity**. This way you can reuse that function every time you want to achieve that aim.

The aim can be very simple such has finding the maximum of a list (function `max`), its length (function `len`) or more complex (finding each unique element and count how many times it occurs).

[1]:
```
def comment_grade(grade):
    if grade < 5:
        return('Grade too low')
    else:
        return('Grade high enough')
```

A function can have optional arguments, always indicated after the compulsory arguments.

[2]:
```
def comment_grade(grade, mode='normal'):
    if grade < 5:
        return('Grade too low')
    elif grade > 5:
        if mode == 'normal':
            return('Grade high enough')
        elif mode == 'positive_reinforcement':
            return('Well done, keep going!')
```

To call the function, use its name, with arguments inside parentheses

[5]:
```
comment_grade(6)
```

[5]:
```
'Grade high enough'
```

In many cases, the user is interested in a return value that he will use.

[14]:
```
comment = comment_grade(6)
```

[15]:
```
print(comment)
```

```
Grade high enough
```

A function can also have no arguments at all. And return nothing.

[7]:
```
def say_hello():
    print('hello')
```

[8]:
```
say_hello()
```

```
hello
```

[16]:
```
greeting = say_hello()
```

```
hello
```

[17]:
```
print(greeting)
```

```
None
```

2.1.1 A special Python keyword: None

`None` is a case-sensitive keyword to state that a variable does not have any value (like "null" in other languages). `None` is not the same as `0`, `False`, or an empty string. `None` has a datatype of its own.

[1]:
```
a = None
print(type(a))
```

```
<class 'NoneType'>
```

A function always return something. So if nothing is explicitely returned by the function, the return value is `None`.

2.1.2 Function docstring

When you write a function, you need to document it. This will be used to automatically generate help and documentation, and also help anyone (including yourself at a later date) better understanding what it does.

[ ]:
```
def comment_grade(grade, mode = 'normal'):
    ''' Provide a feedback according to the grade value

        Parameters
        ----------
        grade : int
            The amount of distance traveled
        mode : str
            The feedback mode, either "normal" (default) or␣
↪"positive_reinforcement"

        Returns
        -------
        comment : str
            The grade feedback
    '''
    if grade < 5:
        return('Grade too low')
    elif grade > 5:
        if mode == 'normal':
            return('Grade high enough')
        elif mode == 'positive_reinforcement':
            return('Well done, keep going!')
```

Best practice is now to use Python "type hints" for the function arguments and return values. You can remove this information from the documentation since it is redundant. Type hints provide useful information when using IDEs (linting) and allows to check code logic via third party libraries (e.g. `mypy`).

[17]:
```
def comment_grade(grade: int, mode: str = 'normal') -> str :
    ''' Provide a feedbac k according to the grade value

        Parameters
        ----------
        grade
            The amount of distance traveled
        mode
            The feedback mode, either "normal" (default) or␣
↪"positive_reinforcement"

        Returns
        -------
        comment
            The grade feedback
    '''
    if grade < 5:
        return('Grade too low')
```

```
    elif grade > 5:
        if mode == 'normal':
            return('Grade high enough')
        elif mode == 'positive_reinforcement':
            return('Well done, keep going!')
```

[15]:
```
student_results = {
                'John': 3,
                'Mary': 9,
                'Peter': 5
                }
```

[7]:
```
for s_name, s_grade in student_results.items():
    s_feedback = comment_grade(s_grade, mode='positive_reinforcement')
    print('Feedback for {}: {}'.format(s_name, s_feedback))
```

```
Feedback for John: Grade too low
Feedback for Mary: Well done, keep going!
Feedback for Peter: None
```

Using f-strings (only the last line change: note the f in front of the string)

[18]:
```
for s_name, s_grade in student_results.items():
    s_feedback = comment_grade(s_grade, mode='positive_reinforcement')
    print(f'Feedback for {s_name}: {s_feedback}')
```

```
Feedback for John: Grade too low
Feedback for Mary: Well done, keep going!
Feedback for Peter: None
```

2.2 Proper Python code development: use an IDE (e.g. Visual Studio Code)

Jupyter notebooks do not allow to easily debug, reuse and version-control your code. As such it is much better practice to develop functions inside an Integrate Development Environment (IDE) to write the main functions there. You can then import them inside your Jupyter Notebook (and reuse them in other notebooks, in other projects, and with other people with e.g. GitHub).

2.2.1 Functions can be (and often are) imported from modules

[26]:
```
from random import randrange
```

[28]:
```
randrange?
```

[33]:
```
max_grade = 10
```

[38]:
```
random_grade = randrange(0, max_grade+1)
print(random_grade)
```

7

[39]:
```
import random
```

[40]:
```
random_grade = random.randrange(0, max_grade+1)
print(random_grade)
```

4

[4]:
```
import os
```

[ ]:
```
os.getcwd()
```

Remember `PATH` in the Linux lecture ?

[1]:
```
import sys
```

[ ]:
```
sys.path
```

[6]:
```
os.__file__
```

[6]:
```
'/opt/conda/shared/envs/ds38/lib/python3.8/os.py'
```

A module is a python file (ending in `.py`), usually containing python functions

To import functions, simply use `import` followed by the name of your python file without the `.py` extension, e.g. `import grading` if your python file is called `grading.py`.

2.2.2 Exercise (using module in Jupyter notebook)

1. Open a terminal in VS Code (`Terminal` –> `New Terminal`) and create the directory `autograder` in the `python_lecture` directory
2. Create a new file in VS Code (`File` –> `New File`) and copy the definition of the function `comment_grade`
3. Save it in the directory `autograder` with the name `grading.py` (`File` –> `Save As`, then search the `autograder` directory you created, clicking on `..` (parent directory) if required)
4. Go back into the Jupyter notebook and try to import your `grading` module so that to call the function `comment_grade` within your notebook. Does it work ?
5. Add the path to your `autograder` directory by appending it to the `sys.path` list.
   Remember that:
   - you can append an element `el` to a list `l` with `l.append(el)`
   - a path is a string (i.e. a sequence of characters) and so should be put in between quotes
6. Try again to import your `grading` module
7. Print the help of your function (`help(fun)`, or `fun?`, or `Shift + Tab` after having written the function name)
8. Call the function with a grade of your choice

2.2.3 Exercise (using module in VS Code)

1. Create a new file in VS Code (`File` –> `New File`) and copy:
   - the definition of the dictionary `student_results`
   - the code applying the function `comment_grade` on each item of the dictionary
2. Save the in the directory `autograder` with the name `exam1.py` (`File` –> `Save As`, then search the `autograder` directory you created, clicking on `..` (parent directory) if required)
3. Import the `grading` module in the `exam1.py` file you just saved (no need to update sys.path as the current function directory is always on the path contrarily to `bash`)
4. Make sure your code will use the function `comment_grade` of that module (choose between `import grading` or `from grading import comment_grade` and adapt the code)
5. In the terminal type `python` and then the path to your `exam1.py` file for python to run your code (note: this is the same as clicking on the `run` icon (green triangle) on the top right)
6. Can you see something strange in the outputs?

2.2.4 Example of debugging (`Run` –> `Start debugging`)

2.2.5 Exercise (using module in VS Code)

1. Correct the bug and check it works
2. Save the file
3. Import `exam1` inside the Jupyter notebook. What do you notice ?

Your python file can be used in two ways: * Called on the command line with `python` (e.g. `python exam1.py`) * Imported in another file or in a Jupyter notebook to use all the functions defined inside

Careful: when you import a file, all the code inside it will be executed !

Python use the keywork `__name__` to understand how your file was used: * If it was called from the command line, then `__name__` will be automatically set to `__main__` * If it was imported, then `__name__` will be automatically set to your file name (e.g. `exam1`)

To have a part of your file automatically run on the command line, use:

```
if __name__ == "__main__":
    ...
```

2.2.6 Exercise (using module in VS Code)

1. Modify `exam1` so that it only executes the code when it is run from the command line
2. Test by saving it and running it from the command line
3. Test by importing in the Jupyter notebook and observing nothing happened
   - Tip: use the following to re-import a module which has been modified:
     `import importlib importlib.reload(my_module)`

2.2.7 Exceptions and assert statements

Errors are essential to be able to understand why something went wrong. You can throw yourself so called “Exceptions” in your code should something unexpected occurs, such as another module or a user not using your function as intended.

[25]:
```
def comment_grade(grade: int, mode: str = 'normal') -> str :
    ''' Provide a feedback according to the grade value

        Parameters
        ----------
        grade
            The amount of distance traveled
        mode
            The feedback mode, either "normal" (default) or␣
↪"positive_reinforcement"

        Returns
        -------
        comment
            The grade feedback
    '''
    if grade >= 0 and grade < 5:
        return('Grade too low')
    elif grade >= 5 and grade <= 10:
        if mode == 'normal':
            return('Grade high enough')
        elif mode == 'positive_reinforcement':
            return('Well done, keep going!')
        else:
            raise ValueError('The mode should be "normal" or␣
↪"positive_reinforcement"')
```

[27]:
```
comment_grade(7, mode="tough")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-27-61b62a4c2f2b> in <module>
----> 1 comment_grade(7, mode="tough")

<ipython-input-25-e9aa7469f387> in comment_grade(grade, mode)
     22             return('Well done, keep going!')
     23         else:
---> 24             raise ValueError('The mode should be "normal" or␣
 ↪"positive_reinforcement"')

ValueError: The mode should be "normal" or "positive_reinforcement"
```

For a list of Python Exceptions, please see here

Assert statements are particularly useful to test the internal logic of your code, to check that impossible events indeed never happen.

[38]:
```
def comment_grade(grade: int, mode: str = 'normal') -> str :
    ''' Provide a feedback according to the grade value

        Parameters
        ----------
        grade
            The amount of distance traveled
        mode
            The feedback mode, either "normal" (default) or␣
↪"positive_reinforcement"

        Returns
        -------
        comment
            The grade feedback
    '''
    if grade >= 0 and grade < 5:
        return('Grade too low')
    elif grade > 5 and grade <= 10:
        if mode == 'normal':
            return('Grade high enough')
        elif mode == 'positive_reinforcement':
            return('Well done, keep going!')
        else:
            raise ValueError('The mode should be "normal" or␣
↪"positive_reinforcement"')
    else:
        assert (grade < 0 or grade > 10), 'INTERNAL BUG: grade is not less than␣
↪0 or greater than 10'
        raise ValueError('EXTERNAL ERROR: The grade entered should be between 0␣
↪and 10')
```

[40]:
```
comment_grade(5)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-40-41e0e863e2d4> in <module>
----> 1 comment_grade(5)

<ipython-input-38-5cfe2777a96e> in comment_grade(grade, mode)
     24             raise ValueError('The mode should be "normal" or␣
 ↪"positive_reinforcement"')
     25     else:
---> 26         assert (grade < 0 or grade > 10), 'INTERNAL BUG: grade is not␣
 ↪less than 0 or greater than 10'
```

```
     27         raise ValueError('EXTERNAL ERROR: The grade entered should be␣
 ↪between 0 and 10')
     28

AssertionError: INTERNAL BUG: grade is not less than 0 or greater than 10
```

It is convenient to define test functions (starting with `test_`) to run a group of assert statements. The `pytest` package is helpful to automatically run this kind of test functions (cf later section).

[41]:
```
def test_comments():
    assert comment_grade(0, mode='normal') == 'Grade too low'
    assert comment_grade(2, mode='normal') == 'Grade too low'
    assert comment_grade(5, mode='normal') == 'Grade high enough'
    assert comment_grade(5, mode='positive_reinforcement') == 'Well done, keep␣
 ↪going!'
    assert comment_grade(10, mode='normal') == 'Grade high enough'
    assert comment_grade(10, mode='positive_reinforcement') == 'Well done, keep␣
 ↪going!'
```

[45]:
```
test_comment_grade()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-45-a3b233d16f3f> in <module>
----> 1 test_comment_grade()

<ipython-input-41-050aabd910cd> in test_comment_grade()
      2     assert comment_grade(0, mode='normal') == 'Grade too low'
      3     assert comment_grade(2, mode='normal') == 'Grade too low'
----> 4     assert comment_grade(5, mode='normal') == 'Grade high enough'
      5     assert comment_grade(5, mode='positive_reinforcement') == 'Well␣
 ↪done, keep going!'
      6     assert comment_grade(10, mode='normal') == 'Grade high enough'

<ipython-input-38-5cfe2777a96e> in comment_grade(grade, mode)
     24             raise ValueError('The mode should be "normal" or␣
 ↪"positive_reinforcement"')
     25     else:
---> 26         assert (grade < 0 or grade > 10), 'INTERNAL BUG: grade is not␣
 ↪less than 0 or greater than 10'
     27         raise ValueError('EXTERNAL ERROR: The grade entered should be␣
 ↪between 0 and 10')
     28

AssertionError: INTERNAL BUG: grade is not less than 0 or greater than 10
```

`pytest` can be used not only to run tests from functions, but also to check results of Numpy docstring examples. In this case, use the `--doctest-modules` flag.

[55]:
```
def comment_grade(grade: int, mode: str = 'normal') -> str :
    ''' Provide a feedback according to the grade value

        Parameters
        ----------
        grade
            The amount of distance traveled
        mode
            The feedback mode, either "normal" (default) or␣
↪"positive_reinforcement"

        Returns
        -------
        comment
            The grade feedback

        Examples
        --------
        >>> comment_grade(6)
        'Grade high enough'

    '''
    if grade >= 0 and grade < 5:
        return('Grade too low')
    elif grade >= 5 and grade <= 10:
        if mode == 'normal':
            return('Grade high enough')
        elif mode == 'positive_reinforcement':
            return('Well done, keep going!')
        else:
            raise ValueError('The mode should be "normal" or␣
↪"positive_reinforcement"')
```

3 Introduction to Python - Part 3

3.1 Object oriented programming (OOP)

Python allows you to implement object oriented programming (OOP). While we can use only a set of data structures and functions to describe the problem we are trying to solve, it can be more natural to describe conceptual entities which have properties (e.g. a student has a first name, last name, average exam grade), and ways of interacting with them (e.g. updating average grade after new exam, providing full name, …).

The properties and ways of interacting are called *data attributes* (often just ***attributes***) and *method attributes* (often just ***methods***) respectively.

This approach tends to better match our mental representation of the world, and thus to help thinking more clearly about the code.

The new conceptual type we define is called a class. Creating an object of this class, (e.g. a student with first name `John`, surname `Doe` and average grade `8`) is also called *creating an instance* of that class, or *instantiating that class*.

Let's see an example of class definition. Creating an object often requires providing essential attributes (e.g. name and first name for a student). In Python, this is done with the special function `__init__`, and the special keyword `self`.

[40]:
```
### Skeleton version without type hints and numpydoc
class Student:

    def __init__(self, first_name, last_name, avg_grade=None):
        self.first_name = first_name
        self.last_name = last_name
        self.avg_grade = avg_grade

    def get_fullname(self):
        return self.first_name + ' ' + self.last_name
```

[41]:
```
### After adding type hints
class Student:

    def __init__(self, first_name: str, last_name: str, avg_grade: float =␣
 ↪None) -> None:
        self.first_name = first_name
        self.last_name = last_name
        self.avg_grade = avg_grade

    def get_fullname(self) -> str:
        return self.first_name + ' ' + self.last_name
```

[1]:
```
### After adding type hints and numpydoc
class Student:
    """
    Named student with yearly grade average.


    Attributes
    ----------
    first_name : str
        Student first name
    last_name : str
        Student last name
    avg_grade : float
        Average grade (default None)
```

```
    Methods
    -------
    get_fullname()
        Get the full student name
    """

    def __init__(self, first_name: str, last_name: str, avg_grade: float =␣
↪None) -> None:
        self.first_name = first_name
        self.last_name = last_name
        self.avg_grade = avg_grade

    def get_fullname(self) -> str:
        return self.first_name + ' ' + self.last_name
```

[2]:
```
student1 = Student('Jane', 'Smith')
student2 = Student('John', 'Doe', 8.5)
```

[3]:
```
print(f'Student 1 full name is {student1.get_fullname()}, and is average grade␣
↪{student1.avg_grade}')
```

```
Student 1 full name is Jane Smith, and is average grade None
```

[4]:
```
print(f'Student 2 full name is {student2.get_fullname()}, and is average grade␣
↪{student2.avg_grade}')
```

```
Student 2 full name is John Doe, and is average grade 8.5
```

[16]:
```
type(student1)
```

[16]:
```
__main__.Student
```

[31]:
```
my_str = 'This is a string'
type(my_str)
```

[31]:
```
str
```

[23]:
```
my_int = 2
type(my_int)
```

[23]:
```
int
```

[33]:
```
dir(student1)
```

[33]:
```
['__class__',
 '__delattr__',
```

```
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'avg_grade',
 'first_name',
 'get_fullname',
 'last_name']
```

[34]:
```
dir(my_int)
```

[34]:
```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
```

```
'__getattribute__',
'__getnewargs__',
'__gt__',
'__hash__',
'__index__',
'__init__',
'__init_subclass__',
'__int__',
'__invert__',
'__le__',
'__lshift__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmod__',
'__rmul__',
'__ror__',
'__round__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__trunc__',
'__xor__',
'as_integer_ratio',
```

```
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
```

3.1.1 How to use specific groups of packages (at given versions) for each project ?

[5]:
```
import numpy as np
```

[6]:
```
np.sqrt(9)
```

[6]:
```
3.0
```

To create an environment and activate it:

- `conda create --name datascience38 python=3.8`
- `conda activate datascience38`

To install packages in an environment (once activated):

- `conda install numpy`

You can use conda to switch between environments:

- `conda activate proj27`
- `conda deactivate`
- `conda activate proj38`
- etc.

To list packages in current environment

- `conda list`

To remove a package, e.g. `numpy`

- `conda remove numpy`

To list environments:

- `conda info --envs`

To remove an environment:

- `conda remove -n datascience38 --all`

Demonstration when calling python interpreter from command line

3.1.2 Exercise (using terminal in VS Code)

We will prepare a conda environment for our next project on the International Space Station (ISS). We will use `python 3.8` in the environment, so we will call it `iss38`.

1. Make sure you have a terminal or start one (`Terminal` –> `New Terminal`)
2. Create a new environment called `iss38` using `python 3.8`
3. Activate that environment
4. List existing packages
5. Install the packages `requests`, `reverse-geocode`, `pylint` and `pytest`
6. List again existing packages to check they were properly installed
7. Change your current VS Code environment to `iss38`
    - Click on the current Python environmnent in the bottom left, then choose the `iss38` environment in the menu appearing on the top

3.1.3 ISS and astronauts

[12]:
```
# Jupyter notebook CELL 1 for exercise
import requests
# Get the names of astronauts on board the ISS
astros_json = requests.get('http://api.open-notify.org/astros.json').json()
astro_names = []
for astro_info in astros_json['people']:
    astro_names.append(astro_info['name'])
# Note: equivalent to astro_names = [astro_info['name'] for astro_info in␣
↪astros_json['people']]
print(astro_names)
```

```
['Sergey Ryzhikov', 'Kate Rubins', 'Sergey Kud-Sverchkov']
```

[13]:
```
# Jupyter notebook CELL 2 for exercise
import time
from datetime import datetime
import reverse_geocode

n_trials = 3
trial_pause_seconds = 5
for i_trial in range(n_trials):
    # Get date and time
    date_time_now = datetime.now()
    time_str = date_time_now.strftime("%d-%b-%Y %H:%M:%S")
    # ISS position
    issnow_json = requests.get('http://api.open-notify.org/iss-now.json').json()
    if issnow_json['message']  == 'success':
        iss_latlong = (issnow_json['iss_position']['latitude'],
                       issnow_json['iss_position']['longitude'])
        # The search method expects a list and returns a list.
```

```
        # Hence we need to coerce a single value latlong to a list with␣
↪[latlong]
        # Afterwards we need to extract the result, itself a single-item list,␣
↪with result_list[0]
        iss_pos_list = reverse_geocode.search([iss_latlong])
        iss_pos = iss_pos_list[0]
        iss_info = f'At {time_str}, the ISS was above {iss_pos["city"]} in␣
↪{iss_pos["country"]}'
        print(iss_info)
        time.sleep(trial_pause_seconds)
```

```
At 11-Nov-2020 16:09:54, the ISS was above Kīlauea in United States
At 11-Nov-2020 16:10:00, the ISS was above Kīlauea in United States
At 11-Nov-2020 16:10:05, the ISS was above Kīlauea in United States
```

3.1.4 Exercise (part 1)

Convert the code above into separate functions

1. Using the terminal in VS Code create a new directory in `python_lecture` called `space`
2. Create a new file in VS Code (File –> New File) and save it in the directory `space` with the name `iss.py`
3. Use the code in the two Jupyter notebook cells above to create two functions in the same `iss.py` module (save that file early to benefit from VS Code syntax highligting and other features)
    1. Copy the import statements found in either cell at the top of your `iss.py` file (all modules imported are by convention at the top of a python file)
    2. Create two functions, each corresponding to a cell
        1. A function `get_astro_names` corresponding to the first cell: the function has no arguments and returns the `astro_names` list
        2. A function `get_iss_positions` corresponding to the second cell: the function has the number of trials and the trial pause as arguments, and returns a list of the `iss_info` strings (generated for all trials).
        Hint: you need to create an empty list, and then append the `iss_info` string to that list at each iteration
4. Raise a `requests.exceptions.RequestException` if the API message is not a success
5. Create two test function with one assert statement each:
    1. One function checking that the `get_astro_names` function called always return a list of length between 3 and 6
    2. One function checking that the `get_iss_positions`function called with three trials lasting 0.5 seconds returns a list of length 3.
6. Run pytest (`pytest iss.py`) and check that all tests run correctly
    1. Make sure you have a terminal or start one (`Terminal` –> `New Terminal`)
    2. Make sure you are in the `space` directory (use `ls` and `cd` to navigate the filesystem)
    3. Make sure you are in the `iss38` conda environment (otherwise run `conda activate iss38`)
    4. Type `pytest iss.py`

3.1.5 Exercise (part 2)

1. Go inside the Jupyter notebook and add the path to your `space` directory to `sys.path` (make sure you run `import sys` first inside your Jupyter notebook)
2. Import your `iss.py` module inside the Jupyter notebook (the module name is your filename without the .py extension, i.e. `iss`)
3. Call the first function to print the list of astronauts
4. Call the second function and save the output in a list. Print the list.
5. Go back to VS Code, and add two lines of code to `iss.py` so that it calls one example of each of your functions
6. Test that:
    1. You can run the file on the terminal and it will print the desired output
    2. Importing the same file as a module in Jupyter Notebook will unfortunately also print the ouput

    Tip: use the following to re-import a module which has been modified (replace `my_module` with the name of your module, `iss` for example!)
    `import importlib importlib.reload(my_module)`
7. Go back to VS Code and modify your code so that it the examples are called only if the python file is run on the command line (hint: use `if __name__ == '__main__'`)
8. Test that:
    1. You can run the file on the terminal and it will print the desired output
    2. Importing the same file as a module in Jupyter Notebook will not print anything
9. (If enough time) Create a Numpydoc docstring with correct hint types

3.2 Numerical analysis and plotting with numpy and matplotlib

Numpy is an essential package for scientific computing in python. At its core are n-dimensional arrays, called ndarrays.

ndarrays can be considered as "homogeneous" (same-type elements) nested lists having fixed size in a given dimension.

[7]:
```
import numpy as np
```

[10]:
```
a_1d_list = [2, 3, 5]
a_1d = np.array(a_1d_list)
a_1d
```

[10]:
```
array([2, 3, 5])
```

[49]:
```
b_1d = np.array([4, 6, 7])
b_1d
```

[49]:
```
array([4, 6, 7])
```

Operations are very similar to Matlab (for those familier with it), but **careful with indices and slicing**.

[15]:
```
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
y[0:3]
```

[15]:
```
array([0.        , 0.06342392, 0.12659245])
```

Similar functions to Matlab but different “defaults”

[16]:
```
a_2d = np.array([[2, 3, 5],
                 [8, 12, 16],
                 [18, 22, 26],
                 [27, 31, 40]] )
```

[17]:
```
print(a_2d.shape)
print(np.median(a_2d))
print(np.median(a_2d, axis=1))
```

```
(4, 3)
17.0
[ 3. 12. 22. 31.]
```

You can make up arrays quickly

[18]:
```
zeros_3by2 = np.zeros((3, 2))
ones_4by3 = np.ones((4, 3))
eight_3by2 = np.full((3, 2), 8)
diag_of_5 = np.eye(5)
```

You can index them the usual way as for lists.

[56]:
```
a_2d.dot(eight_3by2)
```

[56]:
```
array([[ 80,  80],
       [288, 288],
       [528, 528],
       [784, 784]])
```

[57]:
```
print(a_2d)
```

```
[[ 2  3  5]
 [ 8 12 16]
 [18 22 26]
 [27 31 40]]
```

[58]:
```
print(a_2d[1:4, :2])
```

```
[[ 8 12]
 [18 22]
 [27 31]]
```

Careful of changes of dimensions. Use slicing to keep dimensions.

[59]:
```
a_2d
```

[59]:
```
array([[ 2,  3,  5],
       [ 8, 12, 16],
       [18, 22, 26],
       [27, 31, 40]])
```

[60]:
```
a_2d_row2 = a_2d[1, :2]
print('fResult of shape {a_2d_row2.shape}: {a_2d_row2}')
```

```
Result of shape (2,): [ 8 12]
```

[61]:
```
a_2d_submat = a_2d[1:2, :2]
print('Result of shape {a_2d_submat.shape}: {a_2d_submat}')
```

```
Result of shape (1, 2): [[ 8 12]]
```

You can create boolean arrays for masking

[19]:
```
a_2d[np.logical_and(a_2d>5, a_2d<20)] = 0
a_2d
```

[19]:
```
array([[ 2,  3,  5],
       [ 0,  0,  0],
       [ 0, 22, 26],
       [27, 31, 40]])
```

And do typical operations. But careful of difference with Matlab: in Python `*` is element-wise multiplication. You need to use `dot` method for matrix dot product. Use `T` method for transpose.

[20]:
```
a_2d * ones_4by3
```

[20]:
```
array([[ 2.,  3.,  5.],
       [ 0.,  0.,  0.],
       [ 0., 22., 26.],
       [27., 31., 40.]])
```

[64]:
```
a_2d.dot(ones_4by3.T)
```

[64]:
```
array([[10., 10., 10., 10.],
       [ 0.,  0.,  0.,  0.],
       [48., 48., 48., 48.],
       [98., 98., 98., 98.]])
```

3.2.1 matplotlib is a core Python plotting library, working particularly well with numpy.

[21]:
```
from matplotlib import pyplot as plt
%matplotlib inline
```

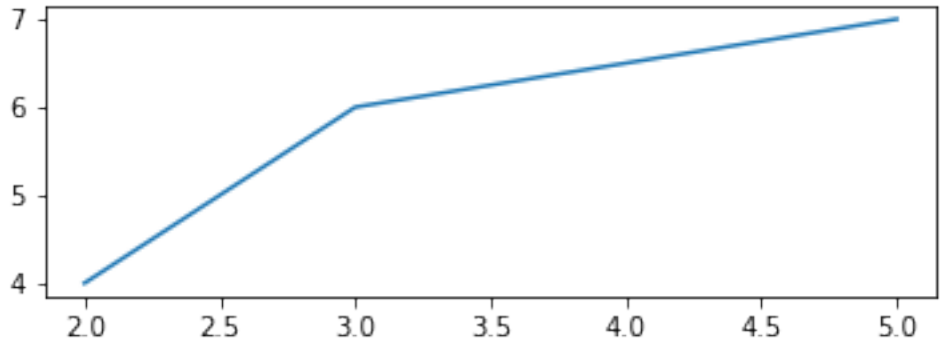
3.2.2 Plot points

[22]:
```
a_1d = np.array([2, 3, 5])
b_1d = np.array([4, 6, 7])
```

Line plot

[24]:
```
plt.figure(figsize=(6,2))
plt.plot(a_1d, b_1d)
```

[24]: [<matplotlib.lines.Line2D at 0x7f884936c9d0>]

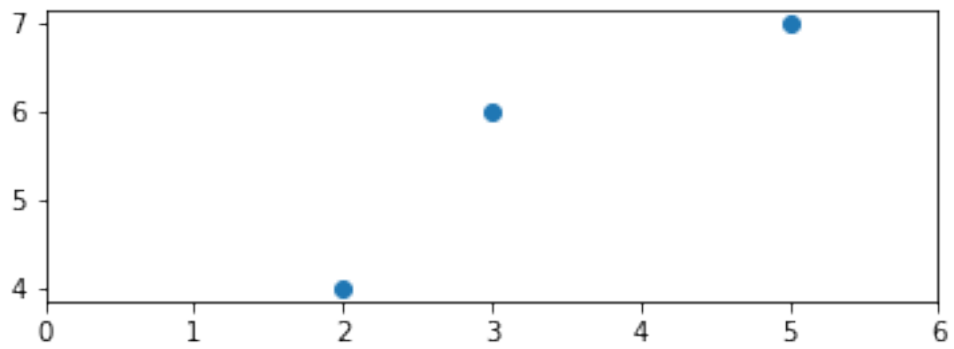


[25]:
```
plt.plot?
```

Scatter plot

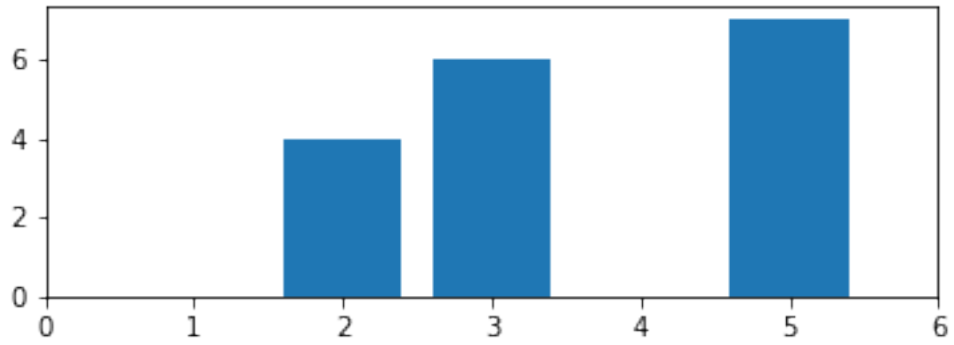
[28]:
```
plt.figure(figsize=(6,2))
plt.scatter(a_1d, b_1d)
plt.xlim((0, 6))
```

[28]: (0.0, 6.0)



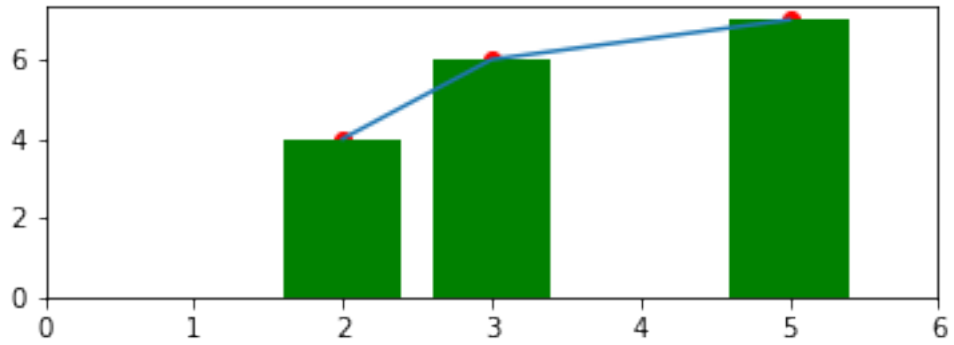
Bar plot

[30]:
```
plt.figure(figsize=(6,2))
plt.bar(a_1d, b_1d)
plt.xlim((0, 6));
```



Overlays

[32]:
```
plt.figure(figsize=(6,2))
plt.plot(a_1d, b_1d)
plt.scatter(a_1d, b_1d, c='r')
plt.bar(a_1d, b_1d, color='g')
plt.xlim((0, 6));
```

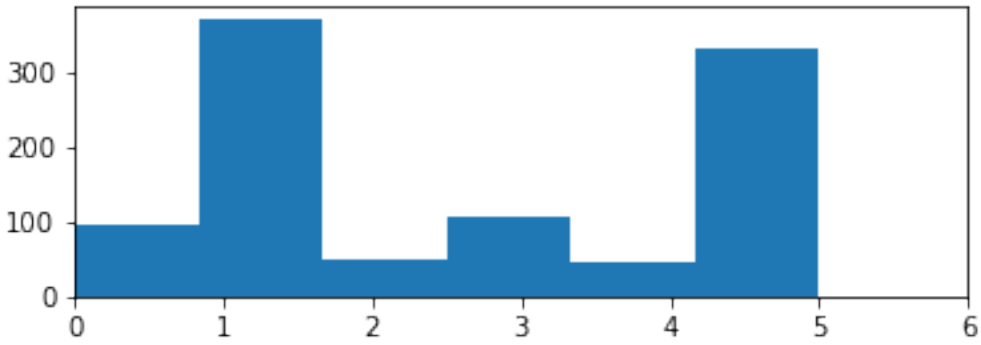


Histogram

[33]:
```
import random
```

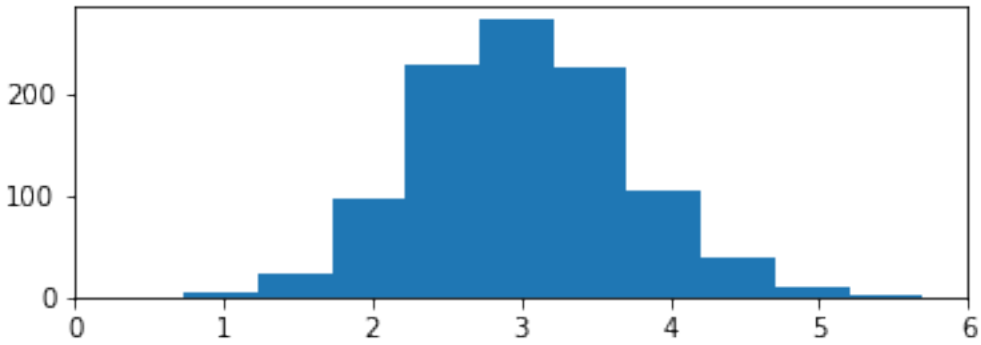
[36]:
```
random.seed(13)
rand6_100 = random.choices(range(6), weights=[0.1, 0.35, 0.05, 0.1, 0.05, 0.
↪35], k=1000)
```

[38]:
```
plt.figure(figsize=(6,2))
plt.hist(rand6_100, bins=6)
plt.xlim((0, 6));
```



[40]:
```
np.random.seed(42)
rand_gauss = np.random.normal(3, 0.7, 1000)
```

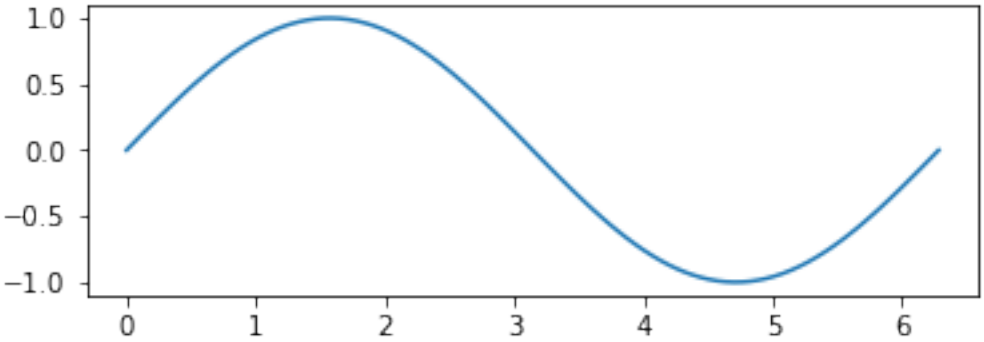
[41]:
```
plt.figure(figsize=(6,2))
plt.hist(rand_gauss)
plt.xlim((0, 6));
```



3.2.3 Plot functions

[42]:
```
x = np.linspace(0, 2*np.pi, 100)
y_sin = np.sin(x)
plt.figure(figsize=(6,2))
plt.plot(x, y_sin)
```

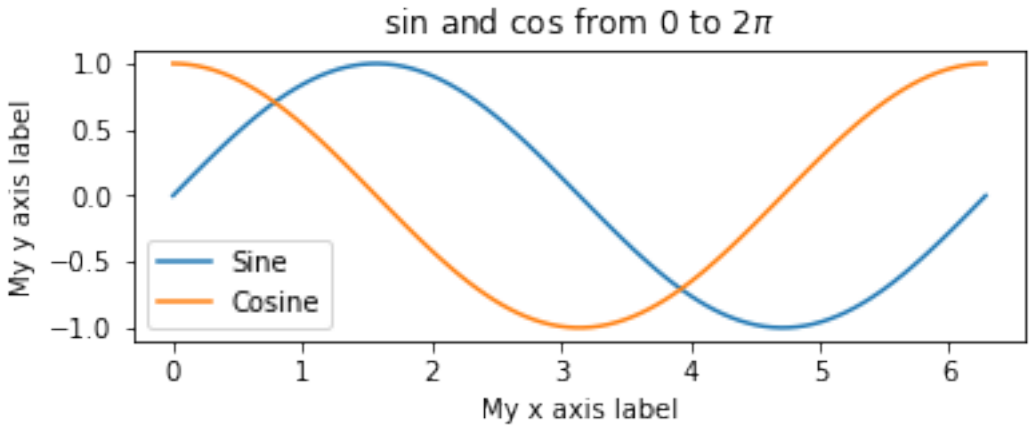
[42]:
```
[<matplotlib.lines.Line2D at 0x7f8848fa6940>]
```



... on the same figure

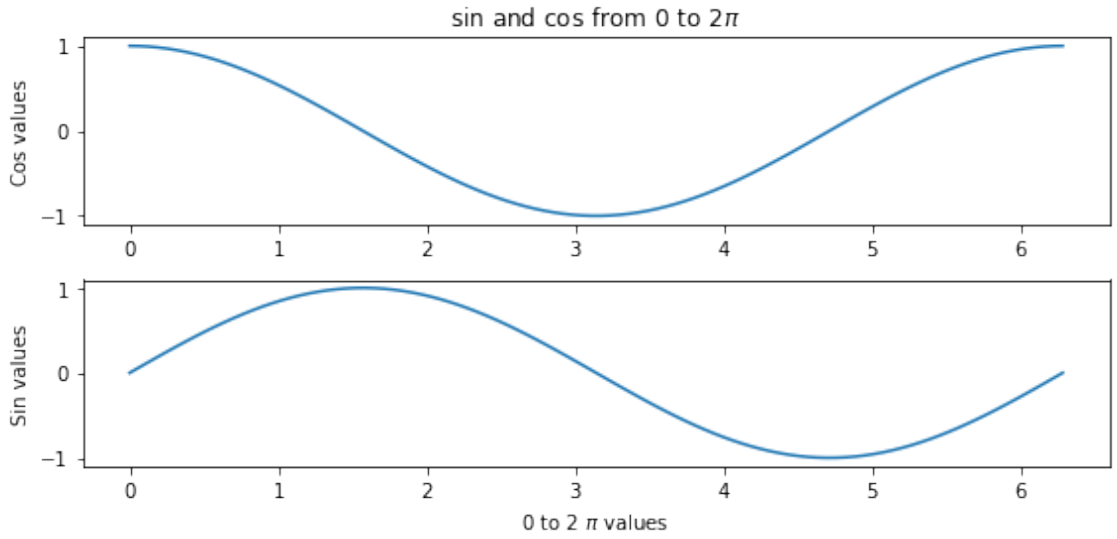
[69]:
```
y_cos = np.cos(x)
# Plot the points using matplotlib
plt.figure(figsize=(6,2))
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('My x axis label')
plt.ylabel('My y axis label')
plt.title('sin and cos from 0 to $2\pi$')
plt.legend(['Sine', 'Cosine'])
```

[69]: <matplotlib.legend.Legend at 0x7f9e704ad7f0>



Subplots

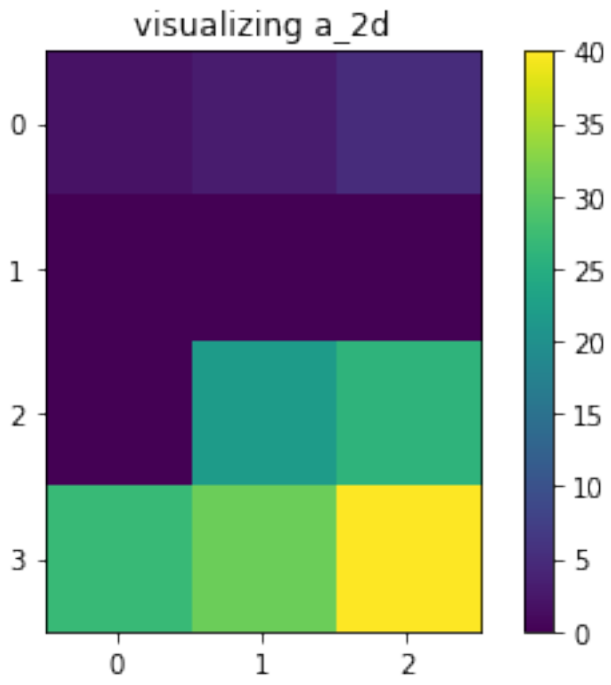
[70]:
```
# Let's plot a total of 2 figures: 1 column and 2 rows
plt.figure(figsize=(8, 4))
n_plots = 2
n_cols = 1
# Plot first figure
plt.subplot(n_plots, n_cols, 1)
plt.plot(x, y_cos)
plt.ylabel('Cos values')
plt.title('sin and cos from 0 to $2\pi$')
plt.subplot(n_plots, n_cols, 2)
plt.plot(x, y_sin)
plt.xlabel('0 to 2 $\pi$ values')
plt.ylabel('Sin values')
plt.tight_layout() # Magic command to clean the overall plot
```



2D plots

[46]:
```
plt.imshow(a_2d)
# Making sure x and y ticks are integers
plt.xticks(np.arange(0, a_2d.shape[1]))
plt.yticks(np.arange(0, a_2d.shape[0]))
# Adding a colorbar
plt.colorbar()
plt.title('visualizing a_2d')
```

[46]:
```
Text(0.5, 1.0, 'visualizing a_2d')
```


visualizing a_2d
0
1
2
3
0
1
2
40
35
30
25
20
15
10
5
0