QLSC 612: Fundamentals of Neuro Data Science

Introduction to Python and virtual environments

Michelle Wang
27 May 2025

Outline

- The Python programming language and some important features
- Python environments
- Running Python code
- Programming in Python
  - Data structures: lists, tuples, sets, dictionaries
  - Functions
  - Classes (briefly)

Resources to get started

Textbooks/tutorials:

- Official Python tutorial (free)
- Think Python, 3rd edition (textbook) (free)
- Python distilled, Python cookbook, Python essential reference (David Beazley)
- Fluent Python (Luciano Ramalho)

Video recording of the 2021 lecture (given by Jacob Sanz-Robinson)

If you already know R or MATLAB:

- Primer on Python for R Users
- An introduction to Python for R Users
- Python for Matlab Users — VOLTTRON 9.0 documentation

The Python programming language

- Created in 1991 by Guido van Rossum
- Free, open, and multiplatform
- Versatile: used in many domains
- Large community and great ecosystem for data science and machine learning
- Simple, easy-to-learn syntax that emphasizes readability

What does this snippet do?

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```

https://pythoninstitute.org/about-python

Features of Python

- **High-level language**: strong abstraction from computer hardware
- **Interpreted language**: does not need to be compiled
- **Dynamically typed**: variables can change types
- **Object-oriented** (more on that later)

The zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Getting started with your Python environment

`conda` is a Python package manager that comes with the Miniconda/Anaconda Python distribution

- Activate the environment created during the installation lab:
  - `conda activate qlsc612` (in a Terminal window)
- You should see your prompt change to have `(qlsc612)` at the beginning

Useful resource: conda cheatsheet

See also: venv (Python built-in) as an alternative to `conda`

- Some systems (e.g., Compute Canada high-performance computers) do not support `conda`

Adding packages to your environment

- `conda install <PACKAGE_NAMES>`
- There are some packages that cannot be installed with `conda`
  - Typically smaller/less popular packages
  - You can use `pip` to install them instead: `pip install <PACKAGE_NAMES>`
- Be careful when mixing `conda install` and `pip install`!
  - In general, install all your `conda`-installable packages first, then `pip install` the rest
  - See this article for details

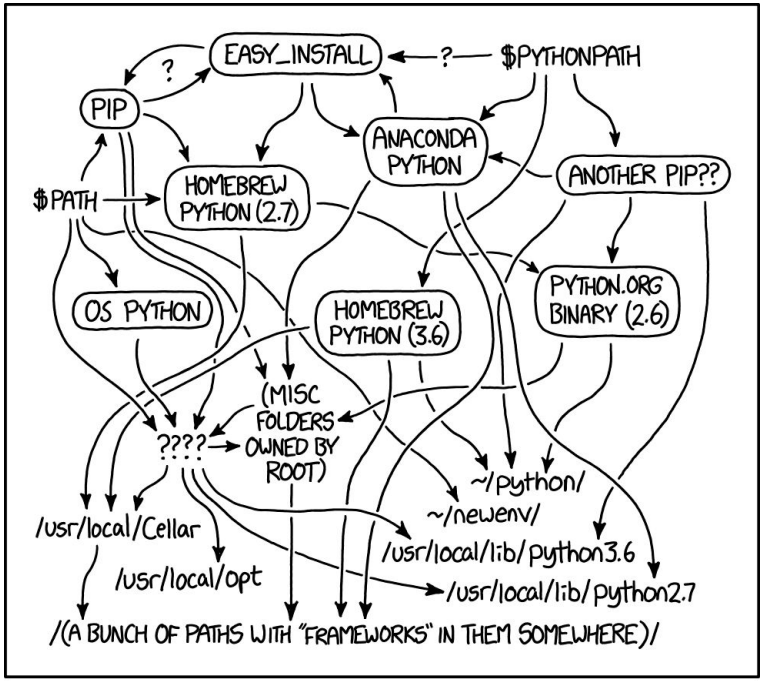
Why use Python environments?

Python Project A requirements

```
python 3.9
numpy 1.20.1
matplotlib 3.3.4
```

Python Project B requirements

```
python 3.7
old_package
numpy ??.??.??
```



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

https://xkcd.com/1987/

Why use Python environments?

It can be difficult to manage different versions of Python and Python packages

- Solution: create a different environment for each project
  - Each environment has its own dependencies
  - Updating one environment does not after the other ones
- **Bonus: increase the reproducibility of your work!**
  - There are ways to help others create a Python environment similar to yours
  - `requirements.txt` file (example)
    - `pip install -r requirements.txt`
  - For `conda` only: `environment.yml` (example)
    - `conda env create -f environment.yml`

What does it mean to “activate a Python environment”?

Let’s look at the `PATH` environment variable before and after activating our environment. In a Terminal window inside VS Code, run:

- `conda deactivate`
- `echo $PATH`
- `which python`
  - This may or may not print anything
- `conda activate qlsc612`
- `echo $PATH`
  - Has anything changed?
- `which python`
  - Has anything changed?

Questions so far?

Next: Running Python code

- Open your VS Code to follow along

Running Python code: from the Terminal

- A Python file is a text file. By convention it should have the `.py` extension
- Let's create and run simple Python script called `hello.py`
- In a Terminal window, type in the command `python` (which points to the Python interpreter) followed by the path to your script
  - E.g., `python hello.py` in this case
  - If you have multiple versions of Python installed and in your `PATH`, you may need to explicitly specify `python3` or `python3.9`
    - Tip: you can use the `which` shell command to check the path of your Python executable
- Alternatively:
  - Add the shebang line `#!/usr/bin/env python` to the top of the file
  - Make the script executable (e.g., `chmod u+x hello.py`) (more on file permissions)
  - Run it as you would run a shell script (e.g., `./hello.py`)

Under the hood: the Python interpreter

- Python is an interpreted language (as are R and MATLAB)
- When you run Python code:
  - The code is translated into **byte code** (`.pyc` files)
    - Low-level set of instructions that can be executed by an interpreter
  - The byte code is executed on a virtual machine (VM) and not a CPU
  - The interpreter checks the validity of variable types and operations (as opposed to having to declare them and having them checked on compilation)
- **Advantage**: given the byte code and the VM are the same version, the byte code can be executed on any platform
- **Disadvantage**: typically slower than compiled languages
  - Compiled languages include C, C++, Java
- More information here

Running Python code: in a Jupyter Notebook

To open the notebook in VS Code, type the following in a **Terminal window**:

- `conda activate qlsc612` # activate the qlsc612 environment
- `code <PATH_TO_NOTEBOOK.ipynb>` # open the notebook in VS Code

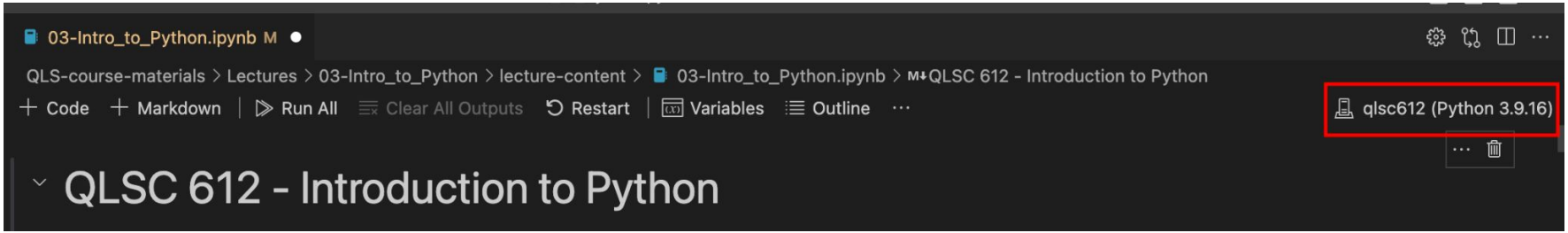
The notebook should now appear in a new VS Code tab or window.

Click on a cell to select it, and press `Ctrl` + `Enter` to execute the code.

If your shell complains that the `code` command cannot be found, you can install it by going to the VS Code Command Palette (`Ctrl`/`Cmd`+`Shift`+`P`) and typing/selecting ***Shell Command: Install 'code' command in PATH***.

Running Python code: in a Jupyter Notebook

- When running a Jupyter notebook in VS Code, you may also need to specify the Python environment (kernel)
- There will be a **Select Kernel** button in the top right corner of the Jupyter notebook, click it and select the one that looks like **qlsc612 (Python 3.9.X) miniconda3/envs/qlsc612/bin/python**
- The button should be updated to read **qlsc612 (Python 3.9.X)**
    - This is the Python environment we have just created for this course: make sure it is the one you are using for later modules.



Some notes on Jupyter Notebooks

- Jupyter notebooks are interactive documents that can combine text elements and code (see also: Project Jupyter and Jupyterlab)
- Running a notebook also requires some external dependencies (e.g., `jupyter`, or `ipykernel` if using VS Code with the Jupyter Notebook extension)

Advantages	Disadvantages

• Interactive!
  ○ Quick iteration cycle
• Code + output + text in the same file
• Great for:
  ○ Exploring data
  ○ Creating plots
  ○ Writing reports
  ○ etc.
	• Can be difficult to version (try to do `git diff` on a notebook)
	• Reproducibility
	  ○ E.g. running cells out of order

The IPython shell

- Interactive Python shell that can be useful for quick checks/debugging
  - Alternative to the regular Python shell (run `python` without any arguments)
- Invoked with the `ipython` command (and requires `ipykernel`/`jupyter`)

```
(qlsc612) Michelles-MBP:04_intro_to_python michellewang$ ipython
Python 3.9.19 | packaged by conda-forge | (main, Mar 20 2024, 12:53:33)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.12.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print('This is the iPython console')
This is the iPython console

In [2]:
```

- If you use `ipython -i <PYTHON_FILE_PATH>` to run a script, at the end of the script it will open an IPython shell will all of your variables/functions there

Questions so far?

Next: Programming in Python

Python super basics

- Python shares some core concepts/elements with other high-level languages:
  - Data types
  - Variables
  - Operators
  - Strings
  - Conditionals
  - Loops
- **These will not be covered explicitly in the lecture**, but we will spend some time on them in exercises so that you can become more familiar with the Python syntax for them
- The `python_basics.ipynb` notebook and/or the `beginners_python_cheat_sheet_pcc_all.pdf` cheat sheet may be useful if you need a refresher

Built-in data structures

- Lists
  - `[1, True, "hello", 4.0]`
- Tuples
  - `(1, True, "hello", 4.0)`
- Sets
  - `{1, True, "hello", 4.0}`
- Dictionaries
  - `{"key1": "value1", 2: 3.0, "4": False}`

Built-in data structures: Lists

- Syntax: `[1, True, "hello", 4.0]`
- Store multiple items in a single variable
- Items in a list can have **different types**
  - Including other lists/data structures
- Lists are **ordered**: each item has a position (index)
  - `[1,2,3]` is not the same as `[3,2,1]`

Built-in data structures: Lists

- List **indexing**: accessing an element in a list

```
>>> my_list = [1, 'two', 3.0, [True, False]]

# first element has index 0
>>> my_list[0]  # get 1
1

# use negative numbers to start at the end
# -1 is the last element
>>> my_list[-1]
[True, False]  # another list
```

Built-in data structures: Lists

- List **slicing**: creating a new list that is a subset of an existing list

```
>>> my_list = [1, 'two', 3.0, [True, False]]

# syntax: start:stop
>>> my_list[1:2]
['two']

# "start" is 0 and "stop" is -1 if omitted
>>> my_list[2:]
[3.0, [True, False]

# syntax: start:stop:step
>>> my_list[::2]
[1, 3.0]
```

Built-in data structures: Lists

- Lists are **mutable**
  - They can be changed without entirely recreating the list
  - Elements can be modified, replaced, added, deleted, order changed

```
>>> my_list = [1, 2, 345, 42]

# note: this does not return anything
# the list is modified in-place
>>> my_list.append("hello")

# my_list is changed
>>> print(my_list)
[1, 2, 345, 42, 'hello']
```

Built-in data structures: Lists

- Be careful when assigning a list to multiple variables!
- Mutability can create unintended consequences/bugs!

```
listA = [0]
print(f"listA, before append: {listA}")
listB = listA
print(f"listB, before append: {listB}")
listA.append(1)
print(f"listA, after append: {listA}")

# what are the contents of listB now?
# visualize in https://pythontutor.com/render.html
```

Built-in data structures: Lists

- How can we copy a list?

```
listA = [0]
print(f"listA, before append: {listA}")

# this creates a copy of the list
# they are no longer the same
listB = listA[:]
print(f"listB, before append: {listB}")

listA.append(1)
print(f"listA, after append: {listA}")
print(f"listB, after append: {listB}")
```

See also: shallow vs deep copies

Built-in data structures: Tuples and sets (briefly)

- Tuples: `(1, True, "hello", 4.0)`
  - Ordered and immutable
  - Allocated more efficiently than list and use less memory
- Sets: `{1, True, "hello", 4.0}`
  - Unordered and mutable
  - Do not allow duplicates

```
# we can typecast between lists, tuples and sets
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
my_set = set(my_tuple)
my_list_from_set = list(my_set)
```

Built-in data structures: Dictionaries

- Syntax: `{"key1": "value1", 2: 3.0, "4": False}`
- Entries as **key-value** pairs. Values can be accessed through their keys
  - They are an implementation of the **hashmap** data structure
- They are **mutable**
- They are **ordered as of Python 3.7** (before 3.7, they were not ordered)
- Duplicate keys are not allowed

Built-in data structures: Dictionaries

- Unlike strings, dictionaries are indexed using **keys**, not the position
  - Keys can be any **hashable immutable** type (strings, integers, tuples, etc.)
  - Values can be anything (including lists or other dictionaries)

```
>>> fruits_available = {
    "apples": 3,
    "oranges": 9,
    "bananas": 12,
    "guanabana": 0,
}

# accessing the value associated with a key
>>> fruits_available['apples']
3
```

Built-in data structures: Dictionaries

- Unlike strings, dictionaries are indexed using **keys**, not the position
  - Keys can be any **hashable immutable** type (strings, integers, tuples, etc.)
  - Values can be anything (including lists or other dictionaries)

```
>>> fruits_available = {
    "apples": 3,
    "oranges": 9,
    "bananas": 12,
    "guanabana": 0,
}

# updating an entry
>>> fruits_available['guanabana'] = 5
```

Built-in data structures: Dictionaries

- Unlike strings, dictionaries are indexed using **keys**, not the position
  - Keys can be any **hashable immutable** type (strings, integers, tuples, etc.)
  - Values can be anything (including lists or other dictionaries)

```
>>> fruits_available = {
    "apples": 3,
    "oranges": 9,
    "bananas": 12,
    "guanabana": 0,
}

# adding a new entry
>>> fruits_available['cherries'] = 10
```

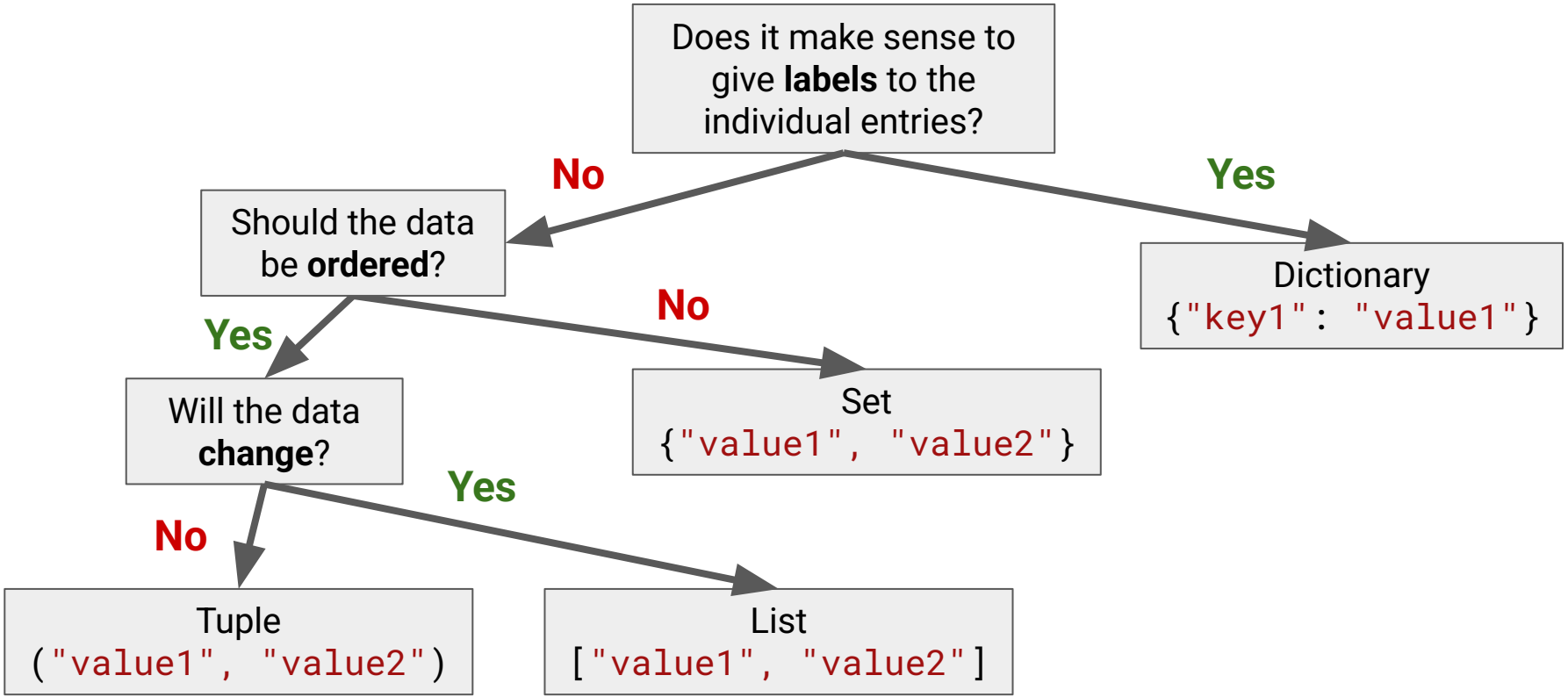
Built-in data structures: Dictionaries

- Unlike strings, dictionaries are indexed using **keys**, not the position
  - Keys can be any **hashable immutable** type (strings, integers, tuples, etc.)
  - Values can be anything (including lists or other dictionaries)

```
>>> fruits_available = {
    "apples": 3,
    "oranges": 9,
    "bananas": 12,
    "guanabana": 0,
}

# deleting an entry (while getting its value)
>>> fruits_available.pop("oranges")
9
```

When to use which data structure?

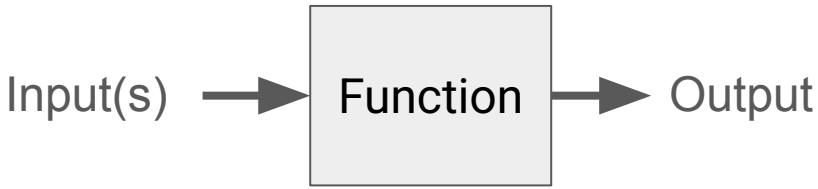


Questions so far?

Next: functions and classes

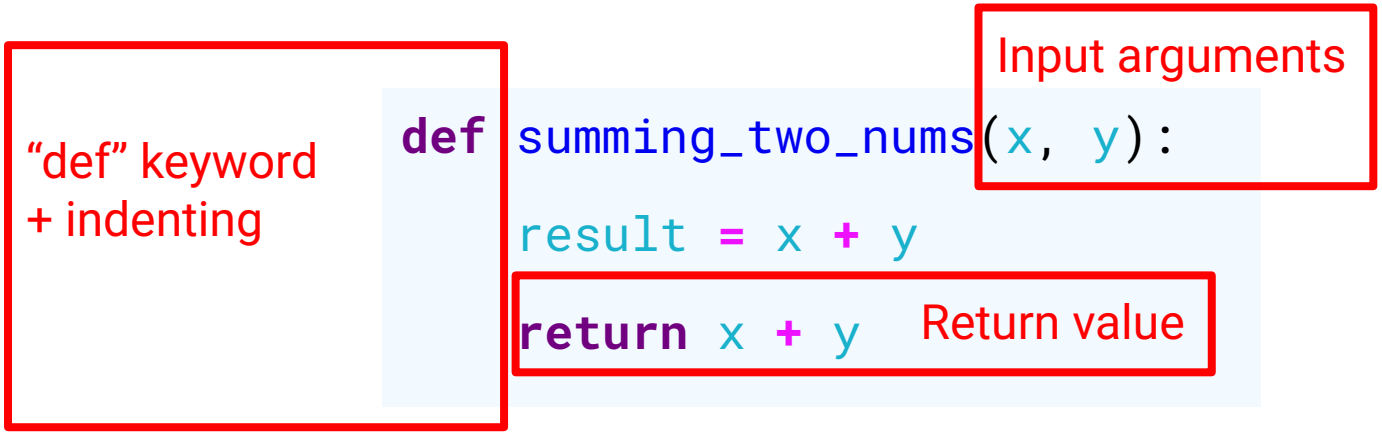
Functions make code reusable

- Block of organized, reusable code
- Typically used to perform a “single” action
- Can be seen as a black box (abstraction)



Function names typically use `snake_case`, same as for variables.

Anatomy of a Python function



Functions and variable scope

- Variables defined within a function are known as **local variables**: they stop existing once the function returns.
- Variables outside of functions are known as **global variables**. They can be accessed inside functions (provided there is no local variable with the same name).

See also: namespaces in Python.

Functions and variable scope

Visualize execution on
https://pythontutor.com/render.html

```
def my_function(my_variable):
    my_variable = "bar"  # local variable with the same name as the global variable
    for _ in range(n):  # accessing a global variable
        print("my_variable inside the function: " + my_variable)


my_variable = "foo"  # global variable
n = 2                # global variable
print("my_variable outside the function: " + my_variable)
my_function(my_variable)
print("my_variable outside the function again: " + my_variable)  # unchanged
```

Passing mutable objects to functions

Visualize execution on
https://pythontutor.com/render.html

```
def change_list(my_list_inside):
    print(f"my_list_inside, before appending: {my_list_inside}")
    my_list_inside.append([1, 2, 3, 4])
    print(f"my_list_inside, after appending: {my_list_inside}")
    return  # note that we are not returning anything


my_list = [10, 20, 30]
print(f"my_list: {my_list}")
change_list(my_list)
print(f"my_list, after change_list() has run: {my_list}")  # changed
```

Functions in your own code

- Do you use functions when coding? Why/why not?

- It is very common to write code by copying previous snippets and slightly modifying them
  - ***Everyone*** has done this
- However, this is usually not the best way to do things. Why?
  - What if I want to modify something?
    - **Maintainability**
  - What if someone else inherits this code or if I come back to it in a year?
    - **Modularity/readability**

Classes and object-oriented programming

- Classes are used to create **user-defined data structures**. They can have their own variables (called **attributes**) and functions (called **methods**)
- Attributes and methods are accessed with the **dot (.) operator**
- An **object** is an instance of a class (class = blueprint) and contains real data

The Python convention is to use `UpperCamelCase` for class names (i.e., first letter of each word capitalized, no symbol separating words)

Classes and object-oriented programming

```
class Dog:
    # Class attribute -- shared between all instances
    species = "Canis familiaris"

    def __init__(self, name, age):  # Specific to instance
        self.name = name
        self.age = age

    def description(self):
        return f"{self.name} is {self.age} years old"

>>> my_dog = Dog("Bonzo", 7)
>>> my_dog.name
>>> my_dog.description()
>>> my_dog.species
```

Classes and object-oriented programming

```
class Dog:
    # Class attribute -- shared between all instances
    species = "Canis familiaris"

    def __init__(self, name, age):  # Specific to instance
        self.name = name
        self.age = age

    def description(self):
        return f"{self.name} is {self.age} years old"
```

- Why would we use classes instead of dictionaries?

Classes and object-oriented programming

- In practice, you probably will do not need to write your own classes when writing code for an analysis
- However, you will still need to use/interact with **classes written by others**, so it is still important to understand them
- Many popular Python packages define classes for others to use, and we will see them often in the rest of the course

Using libraries

- We can import libraries to use functions that other people have written and are inside these libraries
- Some are included by default in Python, and some have to be installed
- Installing libraries depends on your environment manager
  - `conda install [PACKAGE_NAME]`
  - `pip install [PACKAGE_NAME]`
  - (See caveats in earlier slides)

Using libraries

```
import numpy as np  # alias

np.nan  # a constant

np.abs([1, -1])  # a function
```

```
# importing specific objects from the module

from numpy import floor

floor(3.4)  # do not need "np."
```

Practical tips

- Having a code IDE setup can greatly improve your coding experience
  - Syntax highlighting
  - Language server (hover on function name to see signature/docstring, etc.)
  - Type annotations can be helpful
- Writing good code is important, even in research
  - Improve efficiency
  - Reduce mistakes/fix them quickly
- Be critical! And trust that you will improve with experience!