Introduction to Python 2024

Lyda Hill Department of Bioinformatics

Course Overview

Day 1

- Opening Remarks
- Practical Tools for Running Python Software
- Python Data Structures
- Controlling Program Flow
- Functions and Modules
- Dependency Management and Environment Requirements

Day 2

- Introduction to the Python Ecosystem of Libraries
- Reading and Writing Data
- Error Handling in Python
- Introdution to generators and itertools
- Basic Data Operations (Filtering)
- Data Visualization
- Introduction to numpy, skimage and scipy

Why Python?

Why Python?

Modern Software and Scientific Programming.

- **Python's Popularity:**
  - One of the most widely used programming languages globally.
  - Preferred in academia, industry, and research for its simplicity and versatility.
- **Ease of Learning:**
  - Intuitive syntax that resembles human language.
  - Extensive documentation and active community support.
  - Ideal for beginners but powerful enough for experts.

Why Python?

Modern Software and Scientific Programming.

- **Broad Applicability:**
  - Used across multiple domains: web development, data science, automation, scientific research, and more.
  - Extensive standard library and third-party packages available on PyPI (Python Package Index).
- **Rapid Development:**
  - Enables quick prototyping and iterative development.
  - Reduces the time from concept to implementation, especially in research settings.

Why Python?

Modern Software and Scientific Programming.

- **Open Source and Free:**
  - Large ecosystem with a vast number of open-source libraries and tools.
  - Access to data science, machine learning, and artificial intelligence tools.
  - Accessible and affordable for educational and research purposes.
- **Scientific Computing & Research:**
  - Essential for data analysis, simulation, and modeling in scientific research.
  - Libraries like NumPy, SciPy, Pandas, and Matplotlib are standard in the biomedical field.

Why are you interested in Python?

Activity

Install Anaconda

www.anaconda.com/download/

Practical Tools for Running Python Software

Executing Python Software

Practical Tools for Running Python Software

Command Line Execution

Using `python script.py` or running the Python interpreter directly from the terminal.

- **Step 1: Open the Command Line.**
  - Windows: Open Command Prompt (search for `cmd` or `Command Prompt`).
  - macOS/Linux: Open Terminal.
- **Step 2: Navigate to your Working Directory.**

  Use the `cd` command to navigate to the folder where you want to create and execute your Python script.

  ```
  cd path/to/your/folder
  ```

Practical Tools for Running Python Software

Command Line Execution

- **Step 3: Create a Simple Python Script.**

  Use a text editor to create a new Python script called hello.py with the following content.

  ```
  # hello.py
  print("Hello, World!")
  ```

- **Step 4: Execute the Python Script.**

  Run the script using the following command:

  ```
  python hello.py
  ```

Practical Tools for Running Python Software

Command Line Executation

- **Advantages:**
  - Simple and quick for running standalone scripts.
  - Great for automation and batch processing.
  - Efficient for executing complete programs.
- **Disadvantages:**
  - Limited debugging capabilities.
  - No interactivity once the script is running.
  - Less suitable for exploratory analysis or iterative development.

Practical Tools for Running Python Software

Interactive Mode

Entering the Python interpreter by typing `python` in the terminal and running code line-by-line interactively.

- **Step 1: Open the Command Line.**
  - Windows: Open Command Prompt (search for `cmd` or `Command Prompt`).
  - macOS/Linux: Open Terminal.
- **Step 2: Start the Python Interpreter.**
  Enter the following command to start the interactive mode:

```
python
```

  - Once in the interactive mode, your terminal will output information on the `Python` installation.

```
Last login: Thu Oct 10 08:44:41 on ttys000
(base) S155475@SW567797 slides % python
Python 3.9.12 (main, Apr  5 2022, 01:52:34)
```

Practical Tools for Running Python Software

Interactive Mode

Entering the Python interpreter by typing `python` in the terminal and running code line-by-line interactively.

- **Step 3: Test a Simple Python Command.**

  Once the interpreter starts, you can run Python code line-by-line. For example:

  ```
  >>> print("Hello, World!")
  ```

- **Step 4: Define a Simple Variable and Perform a Calculation.**

  Enter commands directly into the interpreter:

  ```
  >>> x = 5
  >>> y = 10
  >>> result = x + y
  >>> print(result)
  ```

- **Step 5: Exit the Python Interpreter.**

Practical Tools for Running Python Software

Interactive Mode

- **Advantages:**
  - Immediate feedback; great for testing small snippets of code.
  - Excellent for learning and experimentation.
  - No need to save or create files for quick code execution.
- **Disadvantages:**
  - Not ideal for running large programs.
  - Requires manually saving work if you want to retain results.
  - Code can’t easily be reused across different projects.

Practical Tools for Running Python Software

JupyterLab Notebooks

A web-based interactive development environment for writing and executing Python code.

- **Step 1: If necessasry, install JupyterLab with pip.**

  ```
  pip install jupyterlab
  ```

- **Step 2: Launch JupyterLab.**

  Open the command line and run the following command to start JupyterLab:

  ```
  jupyter-lab
  ```

  - JupyterLab will launch in the present working directory of your terminal.

- **Step 3: Create a New Notebook.**

  Once JupyterLab is open in your browser, click on the "Python 3" option under the "Notebook" section to create a new Python notebook.

Practical Tools for Running Python Software

JupyterLab Notebooks

- **Step 4: Run Code in Cells.**

  In a Jupyter notebook, code is written in cells. Type the following in the first cell:

  ```
  print("Hello, World!")
  ```

  - To execute the code, press `Shift + Enter` or click the "Run" button.

- **Step 5: Use Markdown for Documentation.**

  You can also create markdown cells for documentation. Switch a cell to markdown mode by selecting ***Markdown*** from the dropdown and typing text for documentation.

Practical Tools for Running Python Software

JupyterLab Notebooks

- **Advantages:**
  - Interactive environment perfect for data science, research, and documentation.
  - Supports code, markdown, and rich media in a single interface.
  - Excellent for visualization and step-by-step execution.
- **Disadvantages:**
  - More resource-intensive compared to command-line execution.
  - Not ideal for large, standalone programs.
  - Can be cumbersome for version control and collaboration.

Practical Tools for Running Python Software

Spyder IDE

An integrated development environment (IDE) specifically tailored for scientific computing with Python.

- **Step 1: Install Spyder (if not already installed).**

  You can install Spyder using pip:

  ```
  pip install spyder
  ```

- **Step 2: Launch Spyder.**

  Open the command line and run the following command to start Spyder:

  ```
  spyder
  ```

Practical Tools for Running Python Software

Spyder IDE

An integrated development environment (IDE) specifically tailored for scientific computing with Python.

- **Step 3: Create or Open a Python Script.**

  Once Spyder is open, you can create a new Python script by going to **File > New File** or open an existing one using **File > Open**. Type the following in the script editor:

  ```
  print("Hello, World!")
  ```

- **Step 4: Execute the Python Script.**

  To run the script, press `F5` or click the "Run" button in the toolbar. The output will appear in the "Console" panel at the bottom.

- **Step 5: Explore Spyder's Features.**

  Spyder offers many useful features, such as variable explorer (view and edit variables), integrated help, and an interactive IPython console.

Practical Tools for Running Python Software

Spyder IDE

- **Advantages:**
  - **Integrated Development Environment:** Includes an interactive Python console, variable explorer, and script editor in a single interface.
  - **Built-in Visualization Tools:** Direct integration with popular libraries like Matplotlib, allowing real-time plotting and visualization.
  - **Interactive Debugging:** Provides robust debugging tools, including breakpoints and stepping through code to help with troubleshooting.
- **Disadvantages:**
  - **Resource-Intensive:** Requires more memory and CPU compared to simpler text editors or the command line.
  - **Not Ideal for Large Projects:** While great for small to medium scripts, Spyder might not be the best choice for managing large-scale software projects.
  - **Less Customizable:** Compared to IDEs like VS Code or PyCharm, Spyder is less customizable.

Python Data Structures

Storing and Manipulating Data in Python

Python Data Types

How are types handled in python?

- Compiled languages like Java, C/C++, etc. are *statically typed*, where types are determined at compile time.

  Python is **dynamically typed**: types are determined at runtime.

```
// C++
float x = 0.0f;
```

```
# python
x = 0.0
```

- Instead of explicit definitions, the **value** determines the type!

```
x = 10      # creates int
y = 10.0    # creates float
```

- Use the `type()` keyword to return the type:

```
print(type(x), type(y))
```

```
<class 'int'> <class 'float'>
```

Python Data Types

How are types handled in python?

- Why is the type `<class 'float'>` and not just `float`?

Every value in Python is an object!

Variables are merely references to these objects.

- When we write `x = 10`, python checks the type of the **value** 10 and creates an instance of the `int()` class. The **variable x** is then assigned to that object.
- If we now write `x = 11`, the object in memory representing "10" gets destroyed and a new `int()` is created somewhere else. **This is different from other languages.**
- This happens because the `int` types are **immutable**, meaning they cannot be changed once created.

Mutability vs. Immutability

- **Mutability** refers to an objects ability to change state (or mutate) once created.
- If an object is **immutable**, its content cannot be changed "in-place". Changes to these objects are handled by creating copies of the object at a new location in memory.
- In Python, the basic data-types such as `int, float, str`, etc are all immutable.

Example: We can use the `id()` keyword to see what happens in memory when we modify an `int` vs. a `list`.

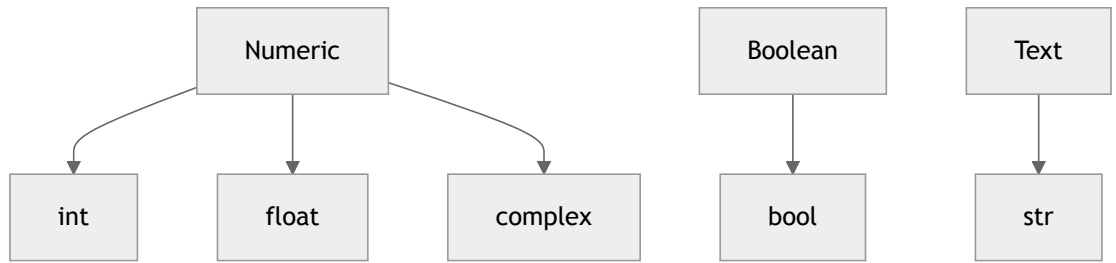
```
x = 10
print("x =", x, "at", id(x))
x = x + 1
print("x =", x, "at", id(x))
y = [1,2,3]
print("y =", y, "at", id(x))
y += [4]
print("y =", y, "at", id(x))
```

```
x = 10 at 140728423185112
x = 11 at 140728423185144
y = [1, 2, 3] at 140728423185144
y = [1, 2, 3, 4] at 140728423185144
```

Note that the address of `x` has changed, while the address of `y` has not. This is because `int` is immutable and `list` is mutable!

Built-in Data Types

Basic data types to store single-values of things like numbers and text.



- Class-nature of types allows us to freely convert between types by **type casting**:

```
x = '5'
y = float(x)
z = bool(y)
print("x =", x, "y =", y, "z =", z)
```

```
x = 5 y = 5.0 z = True
```

- Note the behaviour of `bool()` typecasting! Each type evaluates to `True/False` differently. Here we got `z = True` because `y is not 0.0`. (More on this later!)

Numeric Types

- Numeric types implement the basic arithmetic operations such as `+, -, / and *`.
- Any mixing of `int` and `float` results in type conversion to `float`.

```
print(type(5 + 2))
print(type(5 + 2.0))
```

```
<class 'int'>
<class 'float'>
```

- `float` may be written in scientific notation:

```
x = -1.2e3
print(x)
```

```
-1200.0
```

- Complex numbers are represented in the format `[a] + [b]j`, e.g. the imaginary unit "i" is `1j`. `complex` types store the real and imaginary components of complex numbers:

```
z = 1 + 2j
print(type(z))
print(z.real)
print(z.imag)
```

```
<class 'complex'>
1.0
2.0
```

Strings

- Store words, phrases, paths... Anything represented as text!
- Create using either `' '` or `" "`:

```
a = "Hello!"
print(a)
```

```
Hello!
```

- Strings support some arithmetic operations for manipulation, like `+, *`:

```
b = a + " World!"
c = a*3
print(b)
print(c)
```

```
Hello! World!
Hello!Hello!Hello!
```

- Strings are just **Arrays** of single characters. They have a length and you can access its elements:

```
print(len(a)) # length
print(a[1:4]) # substring
```

```
6
ell
```

Strings (cont'd)

- `f-strings` : convenient way of string formatting with variables. Begin the string with `f` flag and use `{}` to insert variables into strings.

```
a, b = (3,5)
s = f"{a} plus {b} is equal to {float(a+b)}"
print(s)
```

```
3 plus 5 is equal to 8.0
```

- The `in` keyword can be used to search for substrings:

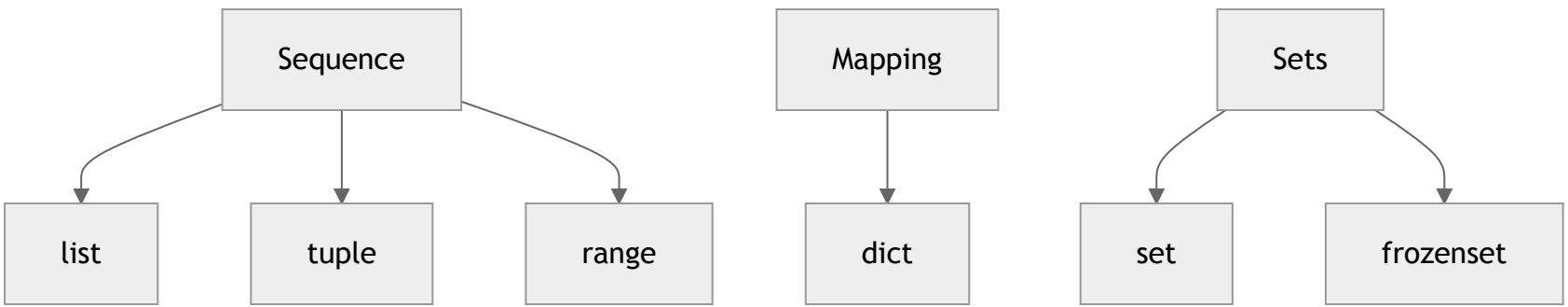
```
print("plus" in s)

if "!!!" not in s:
   print(s + "!!!")
```

```
True
3 plus 5 is equal to 8.0!!!
```

Built-in Data Structures

Useful data types to store collections of data.

- Python has several types which handle multiple elements, each with different behaviour.



- We will focus on `list`, `tuple` and `dict`.

Lists

Essentially Python's answer to what other languages call arrays.

Properties of `list`: **ordered** **mutable** **allow duplicates**

- Create a list using square brackets, ```[ ]```:

```
names = ["susan", "mike", "susan", "joe"] # here we use str, but can be any object!
```

- Or using the constructor directly with some other iterable data type:

```
names = list(("susan", "mike", "susan", "joe")) # creating a list from a tuple
```

- The items in a list are indexed, starting at 0. You can access items in a list by indexing, either relative to the start or the end:

```
print(names[1])
print(names[-1])
```

```
mike
joe
```

Lists (cont'd)

- Recall `list` are mutable; we can freely change its values:

```
names[0] = "kevin"
```

```
['kevin', 'mike', 'susan', 'joe']
```

- Add list items using `.append()` or the `+` operator:

```
names.append("conor")
names += ["felix"] # added item must be list
```

```
['kevin', 'mike', 'susan', 'joe', 'conor', 'felix']
```

- Remove list items using `.remove()`, `.pop()` or the `del` keyword:

```
names.remove("mike")
temp = names.pop(2) # note that pop returns the value!
del names[1]
print(names, temp, "was popped")
```

```
['kevin', 'conor', 'felix'] joe was popped
```

Lists (cont'd)

- Use `.sort()` to sort items alphanumerically:

```
names.sort()
numbers = [5, -1, 10, -3]
numbers.sort()
```

```
names:   ['conor', 'felix', 'kevin']
numbers: [-3, -1, 5, 10]
```

- We will go over `for` loops later, but FYI `list` are `iterable`, which means they can be looped through.

Tuples

Like `list`, but with different purpose and behaviour.

Properties of `tuple`: **ordered** **immutable** **allow duplicates**

- Note the difference here: `tuple` is immutable - its contents can't be changed once created!
- Like list, create a tuple using square brackets, `( )` or `tuple()`:

```
seasons = ("fall", "winter", "spring", "summer")
seasons = tuple(["fall", "winter", "spring", "summer"])
```

- Now if we try and change our `tuple`, e.g. changing "winter" to "summer":

```
seasons[1] = "summer"
```

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

Tuples (Cont'd)

Use `tuple` for collections of data which you are sure should not change!

OK... but why not just use `list` ?

1. Immutability means they are `hashable` : Makes parsing `tuple` slightly faster, more memory efficient.
2. Makes your code clearer and safer for anyone working on it.
3. From the **Zen of Python**: *Explicit is better than implicit. Readability counts.*

- Like `list` , `tuple` can be appended to:

```
seasons = ("fall", "winter", "spring")
print(id(seasons))
seasons += ("summer",)
print("seasons:", seasons)
print(id(seasons))
```

```
2104399491776
seasons: ("fall", "winter", "spring", "summer")
2104399162848
```

- Note that immutability still holds here: the `id` of `seasons` has changed!

Dictionaries (`dict`)

Python's version of maps which store data in **key : value** pairs.

Properties of `dict`: **ordered** **mutable** **NO duplicates!**

- NO duplicates in the sense that a `dict` can't have duplicate `keys`.
- Create `dict` using `{ }` in `{key: value}` format, or using the constructor (not common):

```
employee = {
   "name": "Clara",
   "age": 32,
   "id": 12345
}
employee = dict(name = "Clara", age = 32, id = 12345)
```

- Access `dict` items using `mydict[key]` format:

```
print(employee["name"])
```

```
Clara
```

Dictionaries (`dict`)

- To add new items, simply create a new key:

```
employee["salary"] = 1.2e5
```

```
employee: {'name': 'Clara', 'age': 32, 'id': 12345, 'salary': 120000.0}
```

- Using `.keys()` will give you a `dict_keys` object, which you can convert to `list`:

```
print(list(employee.keys()))
```

```
['name', 'age', 'id', 'salary']
```

- `dict` *values* can be any type, even `dict`:

```
company = {
    "name": "Bell Telephones",
    "employee": employee
}
print(company)
```

```
{'name': 'Bell Telephones',
   'employee': {'name': 'Clara', 'age': 32, 'id': 12345, 'salary': 120000.0}
}
```

- `dict` *keys* can be any *hashable* type, i.e. any immutable type, even `tuple`!

Activity

Get practice storing and manipulating patient data in Python data structures.

Controlling Program Flow

Building Complex Logic

Types of Control Structures

Controlling when and what to execute

Four Ways to Control Code

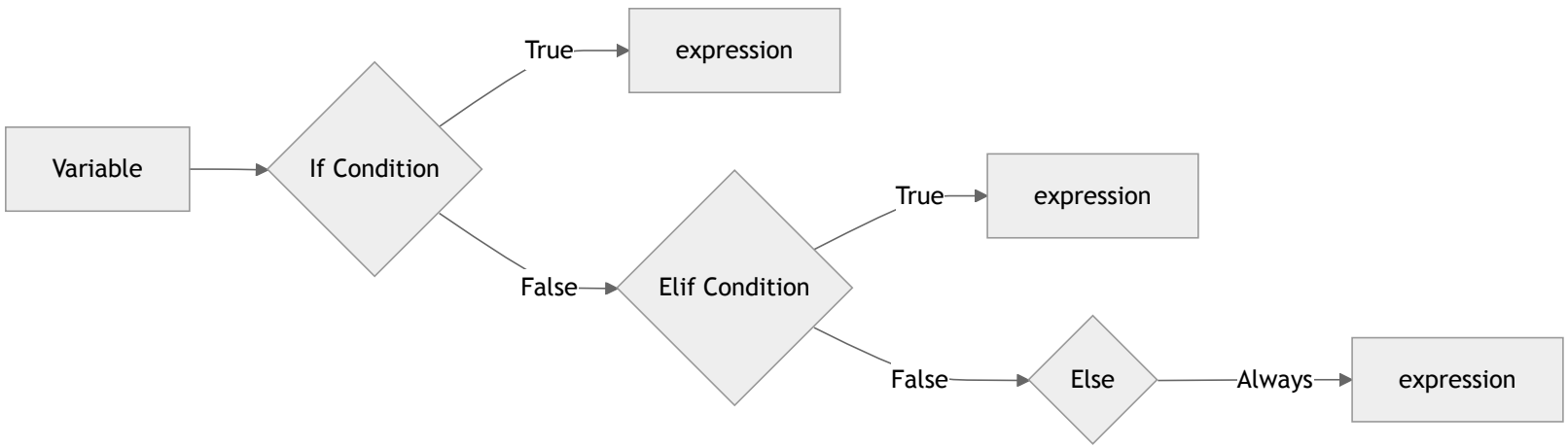
1. Sequential (Default)
   - Code runs one after the other, i.e. top-to-bottom
2. Selection (Conditional)
   - **if** / **if-else** / **if-elif-else**: execute code if a statement is `True` else execute another piece of code
3. Iteration (Loops)
   - **For**: Execute code a finite number of times
   - **While**: Execute code as long as a statement remains `True`
   - Controlling loops: **break**, **continue**, **pass**
4. Try-Except (Try and fail/Gung ho)
   - **try-except-finally**: try to execute code, if fail, try alternative.

Selection (Conditional) Control

if/if-else/if-elif-else

Logic flow

- if-elif-else code is executed in order starting from if, then elif, then else. Think of it as a switch
- You must have **if** first, then any number of **elif**, then **else**
- Condition is any code that generates a boolean output



Selection (Conditional) Control

Indent (Tab) is used to identify code-blocks. The logic is executed in order starting with **if** first, then **elif** etc...

- **if** syntax

```
if condition:
  expression
```

- **if-else** syntax

```
if condition:
  expression
else:
  expression
```

- **if-elif-else** syntax

```
if condition:
  expression
elif condition:
  expression
else:
  expression
```

- You can add an infinite amount of `elif` before the `else`.

Selection (Conditional) Control

if/if-else/if-elif-else

Example

- We use if-elif-else to test whether 3 is divisible by 2 or 3.

```
z = 3

if z % 2 == 0:
    print("z is divisible by 2")
elif z % 3 == 0:
    print("z is divisible by 3")
else:
    print("z is neither divisible by 2 nor by 3")
```

- `%` is the modulo operator which gives the integer remainder of the left number when divided by the right

What is the answer?

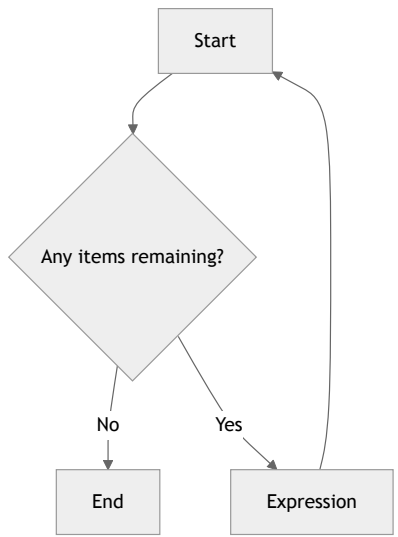
```
  "z is divisible by 3"
```

Iteration (Loops) Control

For and While

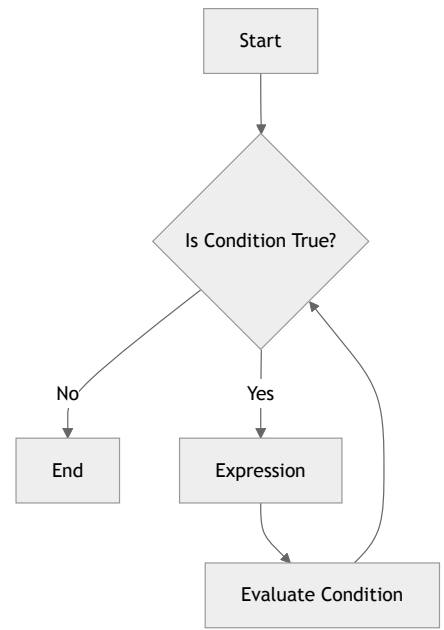
For:

- Do X for every item in a finite iterable



While:

- Do X until a Condition is False



Iteration (Loops) Control

For and While

For can be used to iterate over a list (array) or any other iterator in many different ways

```
x = ['cats','dogs','humans','aliens']
```

1. Directly iterate over the items

```
for item in x:
  print(item)
```

2. Create an index for each item in iterator and iterate

```
for i in range(len(x)):
  item = x[i]
  print(i, item)
```

3. Use `enumerate` to both generate index and iterate item

```
for i, item in enumerate(x):
  print(i, item)
```

Iteration (Loops) Control

For and While

List Comprehension allows us to create a new list from an old one based on some criteria using iteration:

- It is a short-form version of a for loop to make another list

```
names = ["susan", "mike", "susan", "joe"]
upperNames = [name.upper() for name in names]
iNames = [name for name in names if "i" in name]
print(upperNames, iNames)
```

```
['CONOR', 'FELIX', 'KEVIN'] ['felix', 'kevin']
```

- General syntax:

```
newlist = [expression for item in iterable if condition == True]
```

Iteration (Loops) Control

For and While

- Combined with slicing, `::` or indexing `range`, **for** can iterate over a subset of the list

1. using slicing, `start:stop:step`

```
for item in x[::2]: # take every 2nd item including the first.
  print(item)
```

2. using indexing, `range(start:stop:step)`

```
for i in range(0,len(x),2): # take every 2nd item including the first.
  print(i,item)
```

- Indexing, offers the greatest flexibility and control. You can use the index to access data stored in other arrays to perform computation.

Iteration (Loops) Control

For and While

- While loops are used for early termination, primarily if you don’t know how many iterations is needed
  - e.g. searching for `'dogs'` in the list

1. (Using for loop):

```
for i, item in enumerate(x): # take every 2nd item including the first.
  if item == 'dogs':
    # do something
    break # early termination of a finite list
print(i)
```

2. (Using while loop):

```
is_dogs = False # set a condition to test
i = 0 # set index
while is_dogs==False:
  item = x[i]
  is_dogs = item == 'dogs' # re-evaluate the condition
  i=i+1 # update index
print(i)
```

Iteration (Loops) Control

Controlling loop execution

- We can customize the loop to only operate on certain items, skipping or terminating early
- 3 control statements can be used to do this:
  - **Continue**:
    - go direct to next item
  - **Break**:
    - jump out of loop
  - **Pass**:
    - skip code, continue to remainder

```
names = ["Jimmy", "Rose", "Max", "Nina", "Phillip"]

for name in names:
    if len(name) != 4:
        continue

    print(f"Hello, {name}")

    if name == "Nina":
        break
    else:
        pass
    print('after passing: ', name)

print("Done!")
```

```
Hello, Rose
after passing:  Rose
Hello, Nina
Done!
```

Try-Except

Code will fail!

When code fails we can try to catch it and handle exceptions

- `try` lets you test a block, returning an error if it fails
- `except` catches the error and lets you handle it
- `else` executes code when there is no error
- `finally` lets you execute code, regardless of the result of try and except

```
try:
  # Some Code
except:
    # Executed if error occurred in the try block
else:
    # execute if no exception
finally:
    # Some code .....(always executed)
```

Try-Except

Code will fail!

Example: a divide b function

```
# Function to return a/b
def AbyB(a , b):
  try:
      c = ((a+b) // (a-b))
  except ZeroDivisionError:
      print ("a/b result in 0")
  else:
      print (c)

# Testing the above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output:

```
-5.0
a/b result in 0
```

Activity

Write a loop to parse patient metadata and extract desired information

Functions and Modules

Introduction to Functions in Python

What are Functions?

- **What are Functions?**
  - A function is a block of organized, reusable code that performs a specific task.
  - Functions allow you to encapsulate code logic, making it easier to call the same code multiple times without rewriting it.
  - They are defined using the `def` keyword followed by the function name and parentheses `()`.
- **Example of a Simple Function:**

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Introduction to Functions in Python

What are Functions?

- **Benefits of Using Functions:**
  - **Code Reusability:**
    - Functions allow you to write code once and reuse it wherever needed, avoiding repetition.
  - **Modularity:**
    - By breaking down complex problems into smaller functions, you make your code more organized and easier to understand.
  - **Maintainability:**
    - Functions allow you to easily update or debug parts of your code without affecting the rest of the program.

Introduction to Functions in Python

What are Functions?

- **Readability:**
  - Well-named functions make the code more readable by summarizing what specific sections of the code do.
- **Avoiding Global Variables:**
  - Functions help avoid the misuse of global variables by encapsulating variables locally within the function.

Introduction to Functions in Python

Functions with Return Values

- **What are Functions?**
  - In addition to performing tasks, functions can return data back to the caller using the `return` statement.
  - This allows functions to output a result or value that can be used elsewhere in your code.
  - After a `return` statement, the function stops execution and passes the result back.
- **Example of a Function that Returns a Value:**

```
def add(a, b):
    return a + b

result = add(3, 4)
print(result)  # Output will be 7
```

Introduction to Functions in Python

Functions with `*args` in Python

- **What are `*args` in Functions?**
  - `*args` allows you to pass a variable number of positional arguments to a function.
  - The `*` operator collects all extra positional arguments passed to the function and bundles them into a tuple, making it easy to iterate over them or process multiple inputs.
  - This makes functions more flexible, allowing them to accept varying numbers of arguments without needing to be rewritten.

```
def add_all(*args):
    return sum(args)

result = add_all(1, 2, 3, 4)
print(result)  # Output will be 10

result = add_all(1, 2)
print(result)  # Output will be 3
```

Introduction to Functions in Python

Functions with `**kwargs` in Python

- **What are `**kwargs` in Functions?**
  - `**kwargs` allows you to pass a variable number of keyword arguments to a function.
  - The `**` operator collects these arguments into a dictionary.
  - Functions then accept a varying number of keyword arguments without needing predefined parameters.

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)
# Output:
# name: Alice
# age: 30
```

Documenting Your Functions

Documenting Your Functions

Numpydoc

- **What is Numpydoc?**
  - Numpydoc is a docstring format used for documenting Python code, particularly in scientific computing and data science libraries.
  - It's widely used in projects like NumPy, SciPy, and other scientific Python packages.
- **Structure of a Numpydoc Docstring:**
  - Numpydoc follows a specific structure with defined sections to make documentation clear and consistent.
  - Common sections include:
    - **Parameters:** Describes the function's inputs.
    - **Returns:** Explains the output of the function.
    - **Raises:** Lists exceptions that the function can raise.

Documenting Your Functions

Numpydoc

- **Example of a Numpydoc Docstring:**

```
def add_numbers(a, b):
    """ Add two numbers.

    Parameters
    ----------
    a : int
        The first number.
    b : int
        The second number.

    Returns
    -------
    int
        The sum of the two numbers.

    Examples
    --------
    >>> add_numbers(3, 4)
    7
```

Documenting Your Functions

Numpydoc

- **Why Use Numpydoc?**
  - Provides structured, readable documentation that is easy to follow.
  - Helps automate documentation generation for large projects.
  - Widely recognized in the scientific Python community, promoting consistency.

Documenting Your Functions

Inline Type Hinting

- **What is Inline Type Hinting?**
  - Inline type hinting allows you to specify the expected types of function arguments and return values directly in the function signature.
  - Introduced in Python 3.5, type hints improve code readability and help developers understand what types are expected.
  - Type hinting does not enforce type checking but provides a form of documentation for your code.
- **Example of a Function with Type Hinting:**

```
def add_numbers(a: int, b: int) -> int:
    return a + b
```

Documenting Your Functions

Inline Type Hinting

- **Benefits of Using Inline Type Hinting:**
  - **Improved Readability:**
    - Developers can quickly understand what types are expected for arguments and return values.
  - **Better Tooling:**
    - IDEs and code editors can offer better auto-completion, static analysis, and error detection based on type hints.
  - **Facilitates Collaboration:**
    - Team members working on the same codebase can understand function signatures at a glance, improving collaboration and reducing bugs.

Modules

Modules

What are Modules?

- **Definition:**
  - A module is a Python file containing Python code (functions, variables, classes) that can be reused in other programs.
- **Purpose:**
  - Modules allow you to break your code into smaller, reusable, and manageable parts, promoting modular programming.

Modules help organize code and allow for code reuse across multiple programs.

Modules

Built-In Modules

- **The `import` Statement:**

```
import math
print(math.sqrt(16))  # Outputs: 4.0
```

- **Selective Imports (`from module import`):**

```
from math import sqrt
print(sqrt(16))  # Outputs: 4.0
```

- **Aliases (`import module as`):**

```
import numpy as np
print(np.array([1, 2, 3]))
```

Python allows flexible importing: full modules, specific parts, or using aliases for convenience.

Modules

Built-In Modules

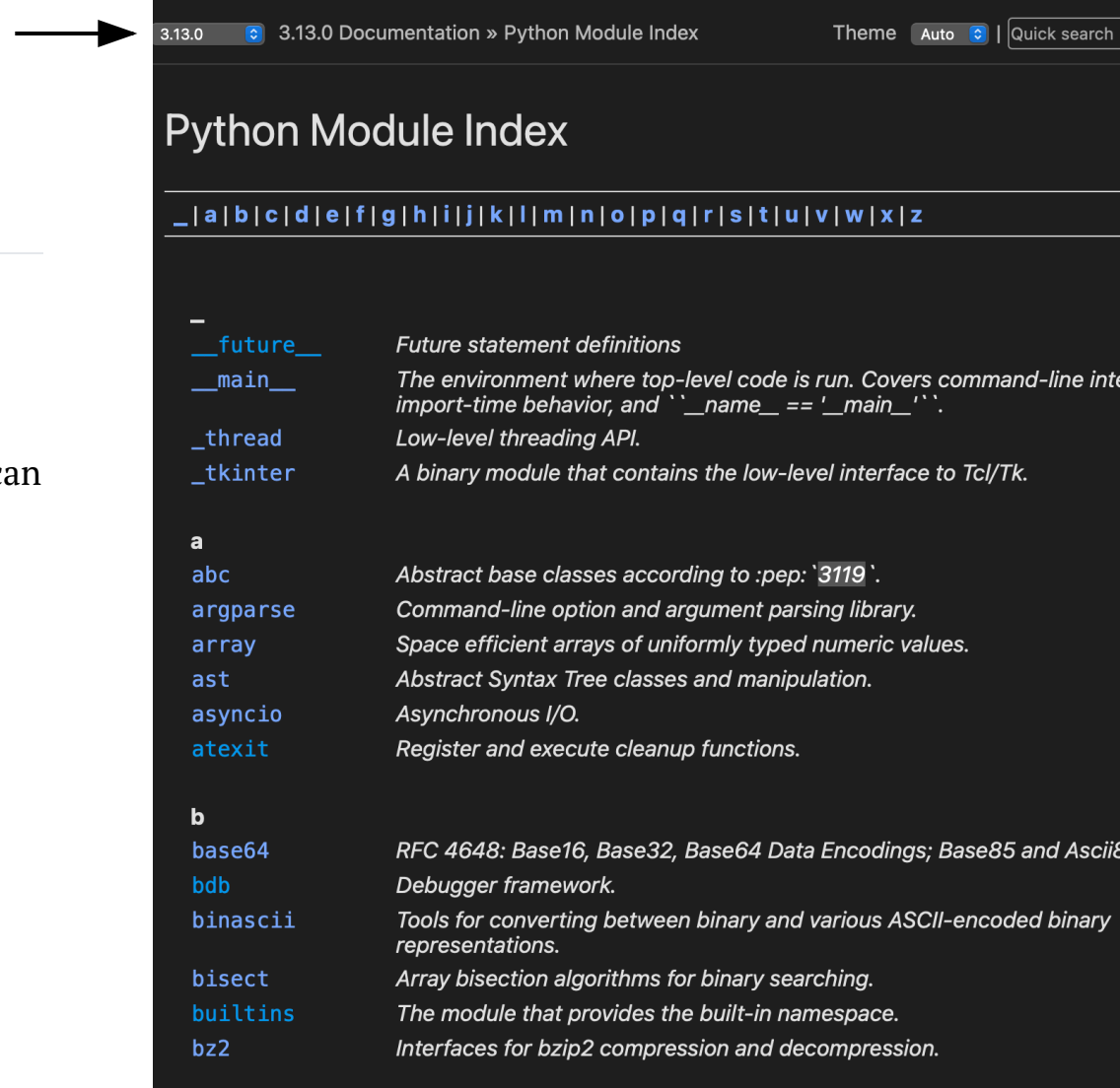
- Python comes with many useful built-in modules. Some examples include:
  - **math:** For mathematical operations.
  - **os:** For interacting with the operating system.
  - **random:** For generating random numbers.
- Example:

```
import random
print(random.randint(1, 10))  # Outputs a random number between 1 and 10
```

Modules

Built-In Modules

- Built-in modules provide essential functionality without requiring external installation.
- A list of built-in modules available to you can be found at https://docs.python.org/3/py-modindex.html



Modules

Creating Custom Modules

- **Creating a Module:**
  - You can create your own module by writing Python code in a `.py` file.
- **Example:**
  - Create a file `my_module.py`:

```
def greet(name):
    return f"Hello, {name}!"
```

  - Use the module in another script:

```
import my_module
print(my_module.greet("Alice"))  # Outputs: Hello, Alice!
```

Custom modules allow you to structure your code and reuse it across different programs.

Modules

The Importance of `__init__.py`

- **What is `__init__.py`?**
  - `__init__.py` is a special Python file used to mark a directory as a Python package.
  - Without this file, Python will not treat the directory as a package, and the modules inside the directory cannot be imported.
  - When a directory contains an `__init__.py` file, you can import modules from the directory like a package.
  - The file can be empty or contain initialization code for the package.

Modules

The Importance of `__init__.py`

- **Basic Example of Package Structure:**

```
my_package/
    __init__.py
    module1.py
    module2.py
```

  - You can import modules from the package:

```
from my_package import module1
```

`__init__.py` is be default empty, but can also be used to intialize package-level variables, import additional submodules, etc.

`__init__.py` is essential for structuring Python packages, enabling module imports, and controlling package initialization behavior.

Modules

Exploring Module Search Path (`sys.path`)

- **How Python Finds Modules:**
  - Python searches for modules using the paths listed in `sys.path` .
- **Checking `sys.path` :**

```
import sys
print(sys.path)
```

- **Modifying `sys.path` :**
  - You can add directories to `sys.path` to include custom modules from other locations:

```
sys.path.append('/path/to/custom/modules')
```

Python's module search path (`sys.path`) determines where Python looks for modules and can be modified to include custom directories.

Modules

Third-Party Modules and PyPI

- **Installing Modules via `pip`:**
  - You can install external Python packages using `pip`:

    ```
    pip install requests
    ```

- **Popular Third-Party Libraries:**
  - `requests`: For making HTTP requests.
  - `numpy`: For numerical computing.
  - `pandas`: For data analysis.

- **PyPI (Python Package Index):**
  - PyPI is the repository for sharing and downloading Python packages.

Third-party modules from PyPI extend Python's functionality, providing solutions for many domains.

Modules

Understanding `if __name__ == "__main__"`

- **What is `__name__`?**
  - In Python, `__name__` is a special built-in variable that represents the name of the current module.
  - When a Python file is run directly, `__name__` is set to `"__main__"`. However, when a file is imported as a module, `__name__` is set to the module's name instead.
- **Why use `if __name__ == "__main__":`?**
  - This statement ensures that a block of code is only executed when the script is run directly, not when it's imported as a module in another script.
  - It is commonly used to encapsulate the "entry point" of a Python script.

Modules

Understanding `if __name__ == "__main__"`

- **Example:**

```
# myscript.py
def main():
    print(5+5)

if __name__ == "__main__":
    main()
```

- **Behavior:**
  - **When run directly:** The `main()` function will execute.
  - **When imported by another file:** The `main()` function will not automatically execute.

It allows for reusable code by preventing specific parts of a script from running when the script is imported as a module elsewhere.

Modules

Benefits of Using Modules

- **Code Reusability:**
  - Reuse code across multiple programs without rewriting it.
- **Organized Codebase:**
  - Keep code clean and organized by dividing it into smaller modules.
- **Maintainability:**
  - Easier to maintain and update your codebase with modular components.

Modules improve code organization, reusability, and maintainability, making it easier to manage larger projects.

Activity

Create and Use Your Own Python Module

Activity

Create and Use Your Own Python Module

Goals

- Create a Python module with two simple functions.
- Import the module into another script and use the functions.
- Properly numpydoc and inline type hint your functions.

Activity

Create and Use Your Own Python Module

Steps:

1. **Create a Python Module:**
   - In your working directory, create a new file called `my_module.py`.
   - Add two simple functions to the file:

```python
# my_module.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Activity

Create and Use Your Own Python Module

2. **Write a Script to Use Your Module:**
   - In the same directory, create another Python file called `use_module.py`.
   - Import your module and use its functions:

```python
# use_module.py

import my_module

result1 = my_module.add(5, 3)
result2 = my_module.subtract(10, 4)

print(f"Addition: {result1}")      # Output: Addition: 8
print(f"Subtraction: {result2}")   # Output: Subtraction: 6
```

Activity

Create and Use Your Own Python Module

---

3. **Run the Script:**
   - In the command line or terminal, run the `use_module.py` script:

```
python use_module.py
```

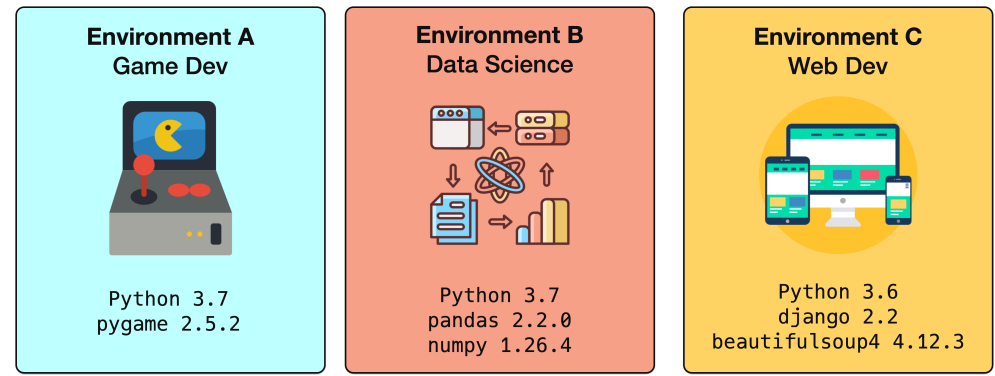
Dependency Management and Environment Requirements

Clean, Reproducible Code

Environment Management with Anaconda

Creating Reproducible Development Environments

- An environment is like a tissue culture flask.
- It guarantees that the software in one project doesn't interfere with another.
- This enables a stable and reproducible space for your code.



Environment Management with Anaconda

Why Does it Matter?

Version Inconsistencies.

- Python libraries and tools are constantly evolving.
- Different projects might require different versions of the same library, leading to conflicts and unexpected behavior.

Reproducibility.

- For scientific computing and data analysis tasks, it’s crucial to reproduce results.
- This is challenging without a consistent environment, especially when sharing work with peers or publishing results.

Environment Management with Anaconda

Why Does it Matter?

Ease of Sharing.

- With a well-managed environment, developers can easily share their projects, ensuring that others can run their code without stumbling upon missing dependencies or version issues.

Isolation.

- Keeping project environments separate ensures that specific dependencies or version requirements of one project don’t interfere with another, leading to cleaner and more stable software.

Environment Management with Anaconda

Miniconda vs Anaconda

Size and Content.

- Anaconda is a large distribution that comes pre-loaded with over 1500 packages tailored for scientific computing, data science, and machine learning.
- Miniconda, on the other hand, is a minimalistic distribution, containing only the package manager (conda) and a minimal set of dependencies.
- Due to its bundled packages, Anaconda requires more disk space upon installation compared to Miniconda.

Flexibility vs. Convenience.

- While Anaconda provides an out-of-the-box solution with a wide array of pre-installed packages, Miniconda offers flexibility by allowing users to install only the packages they need, helping to keep the environment lightweight.

Environment Management with Anaconda

What is a Package?

Software Collection.

- A package is a bundled collection of software tools, libraries, and dependencies that function together to achieve a specific task or set of tasks.

Version Management.

- Each package has specific versioning, allowing users to install, update, or rollback to particular versions as needed, ensuring compatibility and stability in projects.

Dependency Handling.

- When a package is installed in Anaconda, the system automatically manages and installs any required dependencies, ensuring seamless functionality and reducing manual setup efforts.

Environment Management with Anaconda

Creating a New Environment

Create a new environment.

```
conda create --name EnvironmentName
```

Create a new environment with specific Python version

```
conda create --name EnvironmentName python=3.10
```

Create a new environment from a YAML or text file.

```
conda create --name EnvironmentName file=package_contents.yml
conda create --name EnvironmentName file=package_contents.txt
```

Environment Management with Anaconda

Creating New Environments from Text Files

Each package, and all of its dependencies, explicitly imported from a package manager (pip, conda, etc.)

Provides:

- Version Control (e.g., pyserial==3.5)
- Platform Control (e.g., sys_platform == “darwin”)

```
[project]
name = "navigate-micro"
description = "Open source, smart, light-sheet microscopy control software."
authors = [{name = "The Dean Lab, UT Southwestern Medical Center"}]
readme = "README.md"
license = {file = "LICENSE.md"}
dynamic = ["version"]
classifiers = [
    "Intended Audience :: Developers",
    "Operating System :: POSIX :: Linux",
    "Operating System :: MacOS :: MacOS X",
    "Operating System :: Microsoft :: Windows",
    "Programming Language :: Python :: 3.9",
    "Programming Language :: Python :: Implementation :: CPython",
    "Programming Language :: Python :: Implementation :: PyPy",
]

requires-python = ">=3.9.7"
dependencies = [
    'matplotlib-inline==0.1.3',
    'PyYAML==6.0',
    'pyserial==3.5',
    'PIPython==2.6.0.1',
    'nidaqmx==0.5.7',
    'tifffile==2021.11.2',
    'scipy==1.7.3',
    'pyusb==1.2.1',
    'pandas==1.3.5',
    'pandastable==0.12.2.post1',
    'opencv-python==4.5.5.62',
    'numpy==1.22.0; sys_platform != "darwin"',
    'numpy==1.21.6; sys_platform == "darwin"',
    'scikit-image==0.19.1',
    'zarr==2.14.2',
    'fsspec==2022.8.2; sys_platform != "darwin"',
```

Environment Management with Anaconda

Working with Environments

List all environments.

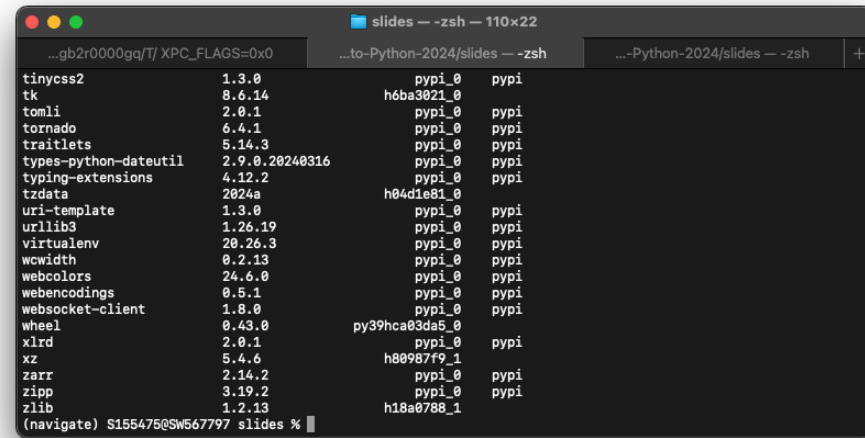
```
conda env list
```

Activate an environment

```
conda activate EnvironmentName
```

List Environment Packages.

```
conda list
```



Adding your environment to a JupyterLab Notebook

1. **Activate Your Environment:**
   - Open your terminal or command prompt and activate the environment you want to add to Jupyter:

     ```
     conda activate environment_name
     ```

2. **Install the `ipykernel` Package (if not already installed):**
   - Ensure that `ipykernel` is installed in your environment so it can be used as a Jupyter kernel:

     ```
     pip install ipykernel
     ```

Adding your environment to a JupyterLab Notebook

1. **Add the Environment as a JupyterLab Kernel:**
   - Use the following command to add your environment as a kernel in Jupyter:

     ```
     python -m ipykernel install --user --name environment_name --display-name "My Environment"
     ```

   - Replace `"environment_name"` with your environment's name, and you can customize the display name shown in Jupyter (e.g., "My Environment").

2. **Verify in JupyterLab:**
   - Open Jupyter Notebook or JupyterLab:

     ```
     jupyter notebook
     ```

   - In a new notebook, go to **Kernel > Change Kernel**, and you should see your environment.

Environment Management with Anaconda

Working with Environments

Important:

- Do not mix package managers (e.g., conda and pip).
- Be judicious and explicit with your dependencies.

Activity

Create a Python Environment and Install Packages

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

Objective:

- Create a new environment with a specific Python version.
- Activate the environment.
- Install dependencies from a text file.

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

Steps:

1. **Create a New Environment with Python 3.9:**
   - Open your terminal or command prompt.
   - Run the following command to create a new environment with Python 3.9:

     ```
     conda create --name myenv python=3.9
     ```

2. **Activate the New Environment:**
   - Once the environment is created, activate it using the following command:

     ```
     conda activate myenv
     ```

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

3. **Create a `requirements.txt` File:**
   - In your working directory, create a new text file called `requirements.txt`.
   - Add a few packages to the file, for example:

     ```
     numpy
     pandas
     matplotlib
     ```

4. **Install Packages from the `requirements.txt` File:**
   - Use the following command to install the packages listed in the `requirements.txt` file:

     ```
     pip install -r requirements.txt
     ```

Activity: Create a Python Environment and Install Packages

Create a Python Environment and Install Packages

5. **Verify the Installation:**
   - After installation, verify that the packages were installed correctly by running:

     ```
     conda list
     ```

6. **Import Dependencies and Execute Logic:**
   - Create a Python script called `plot_image.py`.
   - Import numpy and matplotlib.
   - Create a random 2D matrix that is 512 x 512 pixels.
   - Plot the original image.
   - Manipulate the image (e.g., taket he inverse).
   - Plot the manipulated image.

Working with Data.

Reading and Writing Data Across Operating Systems

Working with Data

Basics of Folder and File Operations

- **Basics of Folder and File Operations**
  - Python provides several modules to interact with the file system, making it easy to navigate folders, list files, and perform operations.
  - **Common Modules:**
    - `os` : Basic operating system functions for working with the file system.
    - `glob` : For pattern matching in filenames.
    - `pathlib` : A modern, object-oriented approach to handle file paths.

Working with Data

Basics of Folder and File Operations

- **Using `os` for Folder/File Operations**
  - **Change Directory:**

```
import os
os.chdir('/path/to/directory')
```

  - **List Files in Directory:**

```
files = os.listdir('/path/to/directory')
print(files)
```

  - **Create a Directory:**

```
os.mkdir('new_folder')
```

`os` works across platforms and provides direct access to system-level commands.

Working with Data

Basics of Folder and File Operations

- **Using `glob` for Pattern Matching**
  - **Basic Pattern Matching:**
    - `glob` allows you to search for files matching a specific pattern, such as all `.txt` files in a folder.

```
import glob
txt_files = glob.glob('*.txt')
print(txt_files)
```

  - **Recursive Search (Python 3.5+):**

```
all_files = glob.glob('**/*.txt', recursive=True)
print(all_files)
```

`glob` permits simple pattern-based searches for files within directories.

50% of time: File and folder manipulation

- **Real data is heterogeneous:** collected in different ways, different data formats
- **Every user is different:** different file naming, different folder stuctures (and different between experiments)
- **Operating systems are different:** different filepath convention
  - Windows: r"C:\Work\Projects\Intro-to-Python-2024\slides"
  - Windows: "C:\\Work\\Projects\\Intro-to-Python-2024\\slides"
  - UNIX: "/home/Work/Projects/Intro-to-Python-2024/slides"
- Most of the time is spent on locating the necessary files, reading and parsing them for the desired data in order to perform analysis:
  - **First 50%** is finding and reading the right files in the right folders into data structures (Built-in Python libraries)
  - **Second 50%** is writing code to extract the data in the right format to use for analysis (Day 1 + libraries)

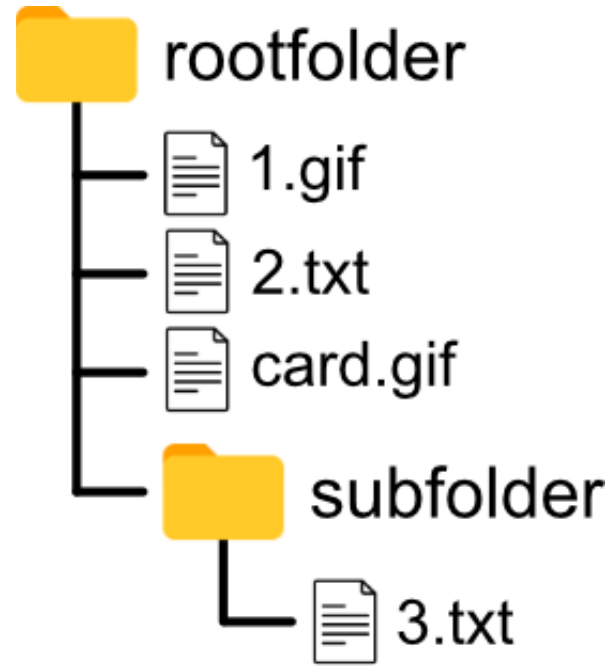
50% of time: File and folder manipulation

`glob` library for file & folder search

- `glob.glob` enables finding files and folders matching a pattern.
- pattern is given by regular expression (regex) e.g. `*` denotes wildcard

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

- `recursive=True` searches also subfolders
- returns list of unordered file or folder paths
- use `sorted()` or `numpy.sort()` to sort



50% of time: File and folder manipulation

`os` library for operating system (OS) operations

- Get current working directory (`os.getcwd`) and change to new working directory (`os.chdir`)

```
import os
cwd = os.getcwd()
os.chdir('/path/to/new/directory')
```

- Create a path from strings (`os.path.join`)

```
os.path.join('prefix/path', 'middle/path', 'suffix/path')
```

- Creating a directory (`os.mkdir`) and all intermediate directories (`os.mkdirs`)

```
os.mkdir('/path/to/new/directory')
os.makedirs('/path/to/new/directory/with/intermediate/directories')
```

- Checking if path is a folder (`os.path.isdir`) or file (`os.path.isfile`)

```
os.path.isdir('/path/to/directory')
os.path.isfile('/path/to/file')
```

Working with Data

Basics of Folder and File Operations

- **Introducing `pathlib`**
  - **Basic Path Creation:**
    - `pathlib` provides an object-oriented approach to handle paths and file operations, and it is cross-platform.

```
from pathlib import Path
path = Path('/path/to/directory')
```

  - **Listing Files in a Directory:**

```
for file in path.iterdir():
    print(file)
```

`pathlib` simplifies path manipulation and handles cross-platform compatibility effortlessly.

Working with Data

Basics of Folder and File Operations

- **Pathlib Advanced Operations**
  - **Check if File or Directory Exists:**

```
if path.exists():
    print("Path exists.")
```

  - **Joining Paths and Accessing File Attributes:**

```
new_path = path / 'new_file.txt'
print(new_path.name)         # Outputs 'new_file.txt'
print(new_path.suffix)       # Outputs '.txt'
print(new_path.parent)       # Outputs '/path/to/directory'
```

`pathlib` is a powerful and user-friendly module for file and folder operations in Python, especially for cross-platform projects.

Activity

Working with Files and Folders in Python

Activity 1

Working with Files and Folders in Python

---

1. **Using `os` for Folder/File Operations**
   - **Change Directory and Create a Folder:**
     - Open a Python script or interactive environment and run the following:

       ```
       import os
       os.chdir('/path/to/your/working/directory')
       os.mkdir('activity_folder')
       ```

   - **List Files in the Current Directory:**
     - Inside the same directory, list all files:

       ```
       files = os.listdir()
       print(files)
       ```

Activity 2

Working with Files and Folders in Python

---

2. **Using `glob` for Pattern Matching**
   - **Find All `.txt` Files in a Directory:**
     - Inside `activity_folder`, create a few `.txt` files and use `glob` to find them:

```
import glob
txt_files = glob.glob('activity_folder/*.txt')
print(txt_files)
```

   - **Recursive Pattern Matching (If using subdirectories):**
     - If you have nested folders, try recursive searching:

```
all_txt_files = glob.glob('activity_folder/**/*.txt', recursive=True)
print(all_txt_files)
```

Activity 3

Working with Files and Folders in Python

---

3. **Using `pathlib` for Cross-Platform Path Operations**
   - **Create a Path Object and List Files in the Directory:**
     - Use `pathlib` to list files and access file properties:

```
from pathlib import Path
path = Path('activity_folder')

for file in path.iterdir():
    print(file.name)   # Print each file name
```

   - **Create a New File Using `pathlib`:**
     - Use `pathlib` to create a new file path and write to it:

```
new_file = path / 'example.txt'
new_file.write_text("Hello, this is a test file.")
```

Activity 4

Working with Files and Folders in Python

---

4. **Challenge: Check File Types and Count Files**
   - **Count Files by Type:**
     - Using `pathlib`, count the number of `.txt` files in `activity_folder`:

```
txt_files_count = sum(1 for f in path.glob('*.txt'))
print(f"Number of .txt files: {txt_files_count}")
```

Working with Data

Introduction to File I/O (Input/Output)

- **Overview:**
  - File handling allows Python programs to interact with files on the system, such as reading data, writing results, and appending new content.
  - Python provides built-in methods for file operations, making it easy to work with text and binary files.
- **Common File Operations:**
  - **Open:** Opens a file for reading or writing.
  - **Read:** Reads data from a file.
  - **Write:** Writes data to a file.
  - **Close:** Closes the file when done.

Working with Data

Introduction to File I/O (Input/Output)

- **File Access Modes**
  - Different modes allow for specific operations:
    - **Read (`r`):** Opens a file for reading. Fails if the file does not exist.
    - **Write (`w`):** Opens a file for writing. Creates a new file or overwrites an existing one.
    - **Append (`a`):** Opens a file for appending. Data is added at the end of the file without overwriting.
    - **Read & Write (`r+`):** Opens a file for both reading and writing.
- **Examples of Opening Files:**

```
file = open("example.txt", "r")  # Open for reading
file = open("example.txt", "w")  # Open for writing (overwrites)
file = open("example.txt", "a")  # Open for appending
```

Working with Data

Introduction to File I/O (Input/Output)

- **Reading Files**
  - **Read Entire File (`read`):** Reads the entire content as a single string.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

  - **Read Lines (`readlines`):** Reads all lines and returns them as a list.

```
with open("example.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

  - **Read Line-by-Line (`for` loop):**

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

Working with Data

Introduction to File I/O (Input/Output)

- **Writing Files**
  - **Write Text (`write`):** Writes a string to a file.

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

  - **Write Multiple Lines (`writelines`):** Writes a list of strings to a file.

```
lines = ["Hello, World!\n", "This is a test.\n"]
with open("example.txt", "w") as file:
    file.writelines(lines)
```

Working with Data

Introduction to File I/O (Input/Output)

- **Appending Text to Files**
  - **Append Mode (`a`):** Adds new content to the end of the file without overwriting existing content.

```
with open("example.txt", "a") as file:
    file.write("Additional line of text.\n")
```

Working with Data

Introduction to File I/O (Input/Output)

- **Best Practice: Using Context Managers**
  - Using a `with` statement automatically handles file closing after operations, even if an error occurs.
  - Example:

```
with open("example.txt", "r") as file:
    content = file.read()  # No need to explicitly close
```

- **Advantages of Context Managers:**
  - Ensures file resources are properly released.
  - Reduces risk of file corruption or data loss.
  - Cleaner code with no need for `file.close()`.

Activity

Introduction to File I/O (Input/Output)

Activity 5: Reading and Counting Lines

Introduction to File I/O (Input/Output)

Objective: Read a text file and count the number of lines.

Instructions:

1. Create a text file called `sample.txt` with some sample text (at least 5 lines).
2. Write a Python script to open the file, read it line by line, and count the total number of lines.
3. Print the line count.

Solution:

```
with open("sample.txt", "r") as file:
    line_count = sum(1 for line in file)
print(f"Total lines: {line_count}")
```

Activity 6: Writing and Appending to a File

Introduction to File I/O (Input/Output)

Objective: Write data to a new file, then append additional data.

Instructions:

Create a script that writes a list of apples, bananas, and carrots to a file called `shopping_list.txt` using `writelines()`. Open the same file in append mode and add tomatoes and potatoes to the file. Finally, print the contents of the file to verify the data was added.

Solution:

```
items = ["apples\n", "bananas\n", "carrots\n"]
with open("shopping_list.txt", "w") as file:
    file.writelines(items)

with open("shopping_list.txt", "a") as file:
    file.write("tomatoes\n")
    file.write("potatoes\n")

with open("shopping_list.txt", "r") as file:
    print(file.read())
```

Activity 7: Filtering Lines Based on Keywords

Introduction to File I/O (Input/Output)

Objective: Read a file and print only lines that contain a specific keyword.

Instructions:

1. Create a text file called `log.txt` with at least 10 lines of sample text. Make sure some lines include the word `"ERROR"`.
2. Write a script that opens `log.txt` and reads each line.
3. Print only the lines that contain the word `"ERROR"`.

Solution:

```
with open("log.txt", "r") as file:
    for line in file:
        if "ERROR" in line:
            print(line.strip())
```

Activity 8: Basic Statistics from a File

Introduction to File I/O (Input/Output)

Objective: Read numerical data from a file and calculate the average.

Instructions:

1. Create a file named `numbers.txt` with one integer per line (at least 5 numbers).
2. Write a Python script to open the file, read the numbers, and calculate the average.
3. Print the average value.

Solution:

```
with open("numbers.txt", "r") as file:
    numbers = [int(line.strip()) for line in file]

average = sum(numbers) / len(numbers)
print(f"Average: {average}")
```

Activity 9: Using Context Managers for Multiple Files

Introduction to File I/O (Input/Output)

Objective: Read data from one file and write it to another using a context manager.

Instructions:

1. Create a file called `source.txt` with some text data.
2. Write a script that reads the contents of `source.txt` and writes it to a new file named `destination.txt`.
3. Use a context manager to handle both files.

Solution:

```
with open("source.txt", "r") as source, open("destination.txt", "w") as dest:
    content = source.read()
    dest.write(content)
```

Working with Data

Introduction to Error Handling in Python

Working with Data

Introduction to Error Handling in Python

- **What is Error Handling?**
  - Error handling allows Python programs to gracefully handle unexpected issues.
  - Common file and folder errors include:
    - **FileNotFoundError:** Raised when attempting to access a file that doesn't exist.
    - **PermissionError:** Raised when the program lacks permission to access a file or folder.
- **Using `try` / `except` Statements**
  - `try` / `except` blocks allow you to catch and handle errors without stopping the entire program.

Working with Data

Introduction to Error Handling in Python

- **Basic Syntax of `try` / `except`**
  - Place the code that may cause an error inside the `try` block.
  - Use an `except` block to define how to handle specific errors.
  - Example:

```
try:
    with open("nonexistent_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: File not found.")
```

Working with Data

Introduction to Error Handling in Python

- **Using Multiple `except` Blocks for Specific Errors**
  - You can handle different types of errors separately by using multiple `except` blocks.
  - Example:

```
try:
    with open("file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: The file does not exist.")
except PermissionError:
    print("Error: You do not have permission to access this file.")
except (IsADirectoryError, NotADirectoryError) as e:
    print("You can even combine multiple exceptions and get the error response...")
    print(f"The error is: {e})
```

Working with Data

Introduction to Error Handling in Python

- **Best Practices for Error Handling in File Operations**
  - **Use Context Managers with `try` / `except` :** The `with` statement ensures files are properly closed, even if an error occurs.
  - **Be Specific with Error Types:** Handle specific exceptions to avoid masking unexpected issues.
  - **Log Errors:** Print or log error messages to understand what went wrong and for easier debugging.
- **Example Combining Context Managers and Error Handling:**

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: File not found.")
except PermissionError:
    print("Error: Permission denied.")
```

Activity 10: Handling Missing Files

Introduction to Error Handling in Python

Objective: Use `try` / `except` to handle errors when a file is missing.

Instructions:

1. Write a script that tries to open a file called `data.txt` for reading.
2. If the file does not exist, catch the `FileNotFoundError` and print a message saying `"File not found. Please check the file path."`

Solution:

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found. Please check the file path.")
```

Working with Data

Introduction to Generators

Working with Data

Introduction to Generators

- **Introduction to Generators**
  - Generators are a special type of function that yield values one at a time using the `yield` keyword instead of `return`.
  - Unlike functions that return all values at once, generators produce values on demand, making them memory-efficient for processing large data streams.
- **Benefits of Generators:**
  - **Memory Efficiency:** Generators yield one item at a time, reducing memory use.
  - **Lazy Evaluation:** Generators only produce values when needed, which is ideal for large or infinite data streams.
  - **Simplifies Code:** Generators simplify writing iterators, allowing for more readable and concise code.

Working with Data

Introduction to Generators

- **Creating a Basic Generator with `yield`**
  - Generators are defined similarly to functions but use `yield` to produce a sequence of values.
  - Example:

```
def simple_generator():
    yield "First"
    yield "Second"
    yield "Third"

for value in simple_generator():
    print(value)
# Output:
# First
# Second
# Third
```

Working with Data

Introduction to Generators

- **Use Case: Processing Large Data Streams**
  - Generators are commonly used for processing data in chunks, such as reading large files line by line:

```
def read_large_file(file_path):
    with open(file_path, "r") as file:
        for line in file:
            yield line.strip()

for line in read_large_file("large_file.txt"):
    print(line)  # Processes one line at a time, conserving memory
```

Key Takeaway:

Generators provide memory-efficient data processing for large datasets by yielding items one at a time.

Working with Data

Introduction to Iterators

Working with Data

Introduction to Iterators

- **Introduction to `itertools` for Advanced Iteration**
  - `itertools` is a Python module providing functions for creating efficient iterators.
  - These tools simplify complex data processing tasks, allowing you to work with infinite sequences, chain multiple iterables, filter data, and more.
- **Useful Functions in `itertools`:**
  - `count`: Generates an infinite sequence of numbers.
  - `cycle`: Repeats elements of an iterable indefinitely.
  - `chain`: Combines multiple iterables into one.
  - `islice`: Slices an iterable, allowing you to control start, stop, and step.

Working with Data

Introduction to Iterators

- **Examples of `itertools` Functions**
  - **Counting Sequence with `count`:**

```
from itertools import count

for i in count(10, 2):
    if i > 20:
        break
    print(i)
# Output: 10, 12, 14, 16, 18, 20
```

Working with Data

Introduction to Iterators

- **Cycling Through a Sequence with `cycle`:**

```
from itertools import cycle

counter = 0
for item in cycle(["A", "B", "C"]):
    print(item)
    counter += 1
    if counter == 6:
        break
# Output: A, B, C, A, B, C
```

Working with Data

Introduction to Iterators

- **Chaining with `itertools`**
  - **Chaining Iterables with `chain`:**

```
from itertools import chain

combined = chain([1, 2, 3], ["a", "b", "c"])
for item in combined:
    print(item)
# Output: 1, 2, 3, a, b, c
```

Working with Data

Introduction to Iterators

- **Slicing with `itertools`**
  - **Slicing with `islice`:**

```
from itertools import islice

for number in islice(count(0), 5, 10):
    print(number)
# Output: 5, 6, 7, 8, 9
```

Key Takeaway:

`itertools` offers advanced iteration tools that streamline tasks like counting, cycling, and chaining, enhancing the efficiency of data processing workflows.

Activity 11: Processing Data Streams with Generators and `itertools`

Introduction to Iterators and Generators

Objective: Use a generator to process a simulated data stream and combine it with `itertools` functions to manipulate the data efficiently.

Instructions:

1. Define a generator function called `data_stream` that yields integers from 1 to 100.
2. Use `itertools.islice` to take only a subset of values from the generator (e.g., 10 values starting from the 5th value).
3. Create a cyclic sequence of labels (`"A"` , `"B"` , `"C"`) using `itertools.cycle` . Pair each value from the sliced data with a label from the cycle, so that each item from `data_stream` gets a label.
4. The output should be pairs like `(5, "A")` , `(6, "B")` , `(7, "C")` , continuing with `(8, "A")` and so on, for 10 values.

Activity 11: Processing Data Streams with Generators and `itertools`

Introduction to Iterators and Generators

Solution:

```
from itertools import islice
# Step 1: Define a generator for a data stream
def data_stream():
    for i in range(1, 101):
        yield i
# Step 2: Use itertools.islice to take a subset of values
stream_slice = islice(data_stream(), 4, 14)  # Starts at 5th item and takes 10 items
# Step 3: Use itertools.cycle to create cyclic labels and pair with data
labels = cycle(["A", "B", "C"])
paired_data = zip(stream_slice, labels)
# Step 4: Print the resulting pairs
for item in paired_data:
    print(item)
# Expected Output:
# (5, 'A')
# (6, 'B')
# (7, 'C')
```

Working with Data

Basic Data Operations (Filtering, Sorting, ...)

Working with Data

Basic Data Operations (Filtering, Sorting, …)

- **Introduction to Basic Data Operations**
  - Python’s core data structures (lists, dictionaries, tuples) are powerful tools for data manipulation.
  - Common operations:
    - **Filtering:** Select specific data points that meet a condition.
    - **Sorting:** Arrange data in a particular order.
    - **Basic Statistics:** Perform simple calculations like sums, averages, or counts.
- **Core Data Structures:**
  - **Lists:** Ordered and mutable, useful for storing sequences.
  - **Dictionaries:** Key-value pairs, fast lookup and retrieval.
  - **Tuples:** Ordered and immutable, often used for fixed data collections.

Working with Data

Basic Data Operations (Filtering, Sorting, ...)

- **Filtering Data in Python**
  - Filtering is commonly done with list comprehensions or the `filter()` function.
  - **Using List Comprehension:**

```
numbers = [1, 2, 3, 4, 5, 6]
evens = [n for n in numbers if n % 2 == 0]
print(evens)  # Output: [2, 4, 6]
```

  - **Using `filter()` Function:**

```
def is_even(n):
    return n % 2 == 0

evens = list(filter(is_even, numbers))
print(evens)  # Output: [2, 4, 6]
```

Working with Data

Basic Data Operations (Filtering, Sorting, ...)

- **Filtering Dictionaries:**
  - Filter based on keys or values in a dictionary.

```
scores = {"Alice": 85, "Bob": 90, "Charlie": 75}
passed = {k: v for k, v in scores.items() if v >= 80}
print(passed)  # Output: {'Alice': 85, 'Bob': 90}
```

- **Sorting Data in Python**
  - **Sorting Lists with `sorted()` and `sort()`:**
    - `sorted()` returns a new sorted list, while `sort()` sorts in place.

```
numbers = [3, 1, 4, 1, 5]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 1, 3, 4, 5]
```

Working with Data

Basic Data Operations (Filtering, Sorting, ...)

- **Sorting with Custom Key Functions:**
  - Sort using a key function, such as by length or by specific value.

```
words = ["apple", "banana", "kiwi"]
sorted_words = sorted(words, key=len)
print(sorted_words)  # Output: ['kiwi', 'apple', 'banana']
```

- **Sorting Dictionaries by Values:**
  - Sort dictionary items by values using `sorted()`:

```
scores = {"Alice": 85, "Bob": 90, "Charlie": 75}
sorted_scores = dict(sorted(scores.items(), key=lambda item: item[1], reverse=True))
print(sorted_scores)  # Output: {'Bob': 90, 'Alice': 85, 'Charlie': 75}
```

Working with Data

Basic Data Operations (Filtering, Sorting, ...)

- **Basic Statistics in Python**
  - **Calculating the Sum and Average:**
    - Use `sum()` and `len()` to calculate the total and average.

```
numbers = [3, 7, 2, 9]
total = sum(numbers)
average = total / len(numbers)
print(f"Total: {total}, Average: {average}")
# Output: Total: 21, Average: 5.25
```

  - **Finding Maximum and Minimum Values:**
    - Use `max()` and `min()` to find the largest and smallest values.

```
numbers = [3, 7, 2, 9]
print(f"Max: {max(numbers)}, Min: {min(numbers)}")
# Output: Max: 9, Min: 2
```

Working with Data

Basic Data Operations (Filtering, Sorting, ...)

- **Using the `statistics` Module for Basic Statistics**
  - Python's `statistics` module provides additional functions for descriptive statistics.
  - **Mean, Median, and Mode:**

```python
import statistics

data = [1, 2, 3, 4, 5, 5]
print(f"Mean: {statistics.mean(data)}")       # Output: Mean: 3.33
print(f"Median: {statistics.median(data)}")   # Output: Median: 3.5
print(f"Mode: {statistics.mode(data)}")       # Output: Mode: 5
```

Key Takeaway:

- **Filtering, sorting, and calculating basic statistics** are essential data operations that can be performed efficiently with core Python data structures.

Introduction to the Python Data Analysis Ecosystem

Don’t re-invent the wheel:
find examples and read documentation!

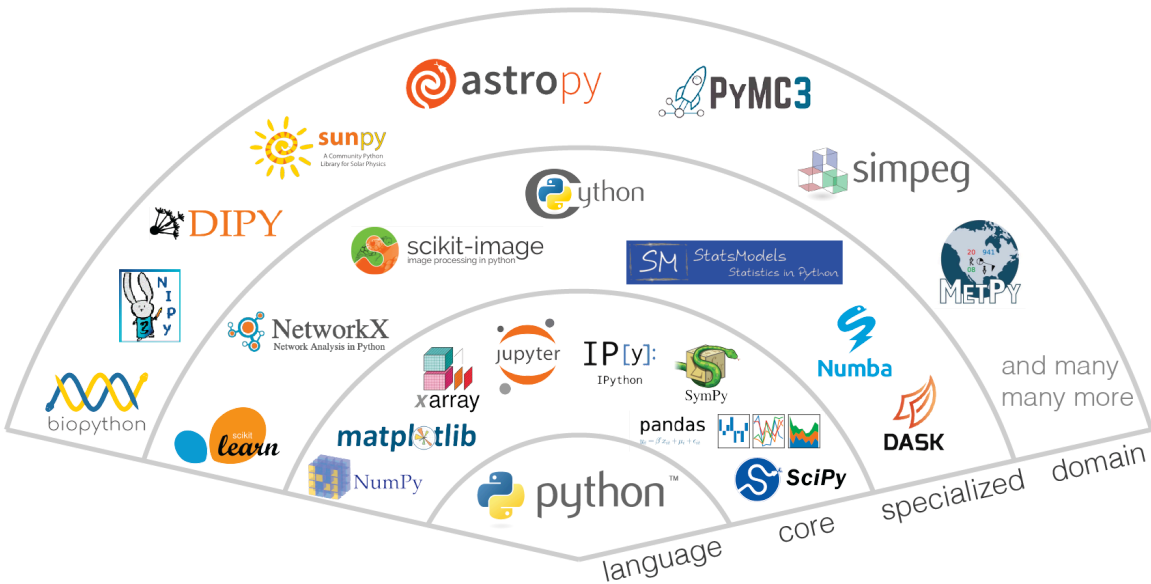
Standard Python has limited functionality and un-optimized for efficiency

- So Far: Python data and control structures
  - We have covered the variable types in python, and how to store them in data structures
  - We have covered how to manipulate variables through control structures
- Today: Using Python in real-world application
  - Basic Python is very limited (We have covered most of it!)
  - Writing code from scratch is suboptimal for real-world applications
    - Low-level compiled languages e.g. C, Fortran more optimized for speed
    - Many functions have already been written and well-maintained (waste of time)
    - Many projects can be solved by mix-and-match
  - **Your objective:** Only write code for data reading and manipulation to use Python libraries and to operate on results

Diverse ecosystem of well-maintained libraries

Different libraries for different uses

- Libraries are 'plugins' providing a set of functions for particular purposes that extend the basic Python.
- Find and install through `conda` or `pip`.
- For custom libraries, installation instructions will be provided in the GitHub repository usually



Essential libraries - 'The Scientific Stack'

You will almost always use these !

These are the most foundational and 'robust' libraries. There is overlap in some functions but generally:

1. **Essential Data Structures**
   - **numpy**: replaces Python list, math library for vectorized N-dimensional maths
2. **Essential Scientific Computing Routines + Common Statistics**
   - **scipy**: hypothesis tests, curve-fitting
3. **Data Visualization**
   - **matplotlib**: can do nearly everything 2D+3D plotting but needs tweaking
   - **seaborn**: wraps matplotlib to provide streamlined plotting of common scientific plots and colormaps
4. **Fundamental Image Processing**
   - **pandas**: Streamlines reading and writing of table formats e.g. .csv, .txt, .tsv, .xlsx. Brings R and SQL table manipulation routines like merge to python.

Essential libraries - 'The Scientific Stack'

You will almost always use these !

5. **Fundamental Tabular Processing**
   - **scikit-image**: Modern image file reading/writing, no support for video like .avi, .mp4, common image processing routines with example code. Includes biological examples.
   - **opencv**: Industry-performant image processing developed in C, now made available for Python. Offers typically faster algorithms but documentation is difficult to understand, few examples, primarily for 2D and general computer vision.
6. **Fundamental Machine Learning**
   - **scikit-learn**: Classical machine learning algorithms with a single function API i.e. algorithm.fit() and algorithm.predict(). Extremely easy to use with many examples.
     - Clustering,
     - Data preprocessing e.g. Standard Scale, one-hot encoding
     - Linear regression models e.g. OLS, Ridge, LASSO, ElasticNet; Random Forests; SVMs

Essential libraries - 'The Scientific Stack'

You will almost always use these !

These are the most foundational and 'robust' libraries. There is overlap in some functions but generally:

- **numpy**:
  - multidimensional arrays that perform fast, vectorized operations (replaces lists)
  - masked arrays, array creation, linear algebra, summary statistics

```
import numpy as np
# python lists do not support vectorized processing
x = ['cats', 'dogs', 'humans', 'aliens']
y = [0,1,2,3,4,5]
# convert to numpy array
x_np = np.array(x); print(x_np.dtype) # this converts to a '<U6', 6-character string
y_np = np.array(y); print(y_np.dtype) # this converts to a np.int32, 32-bit integer
# we can do Boolean operations on all elements simultaneously
x_np_equal_1 = x_np == 'dogs' # return array same size as x_np
# we can compute mean and standard deviation with one function call, and handle NaN's
y_mean = np.nanmean(y_np)
y_std = np.nanstd(y_np)
```

Essential libraries - 'The Scientific Stack'

You will almost always use these !

- **scipy**
  - Basic N-dimensional image processing e.g. correlation, Fourier Transforms, Distance Transforms
  - Linear and non-linear curve-fitting methods
  - Sparse linear algebra
  - Hypothesis Tests

```
import numpy as np
import scipy.stats as spstats

# generate two arrays of random numbers
x = np.random.normal(0,1,100)
y = np.random.uniform(0,1,100)

# Pearson's R
statistic, p_val, conf_internval = spstats.pearsonr(x,y)

# paired t-test
t_test_val, p_val, df, conf_interval = spstats.ttest(x,y)
```

Essential libraries - 'The Scientific Stack'

You will almost always use these !

- **matplotlib**
  - THE plotting library (comparable to ggplot in R)
  - fully customizable plotting (require tweaking to get nice figures)

```
import numpy as np
import pylab as plt # pylab is a shortcut for matplotlib.pylab

# generate two arrays of random numbers
x = np.random.normal(0,1,100)
y = np.random.uniform(0,1,100)
# create a figure and plot x vs y, layering different commands
plt.figure(figsize=(8,8))
plt.title('x vs y', fontname='Arial', fontsize=24)
plt.plot(x,y, 'o', ms=5, mec='k', mfc='w') # plot as white circles, black borders, markersize=5pt
# further customizing plot appearance
plt.xlabel('x-axis', fontsize=18, fontname='Arial')
plt.ylabel('y-axis', fontsize=18, fontname='Arial')
plt.tick_params(right=True, length=10)
plt.savefig('myfigure.svg', dpi=300, bbox_inches='tight') # save as vector format, 300 dpi
plt.show() # tells python to display the figure
```

Essential libraries - 'The Scientific Stack'

You will almost always use these !

- **pandas**
  - THE data table library (comparable to R and SQL)
  - execute manipulations specifically geared for tables
  - interoperates with `numpy`, some libraries e.g. `seaborn` works best on tables.

```
import numpy as np
import pandas as pd

# stack the two random number arrays into a matrix
tab_xy = np.vstack([x,y]).T

# convert to pandas table
pandas_tab_xy = pd.DataFrame(tab_xy, columns=['x','y'], index=None)
# save pandas table
pandas_tab_xy.to_csv('my_table.csv', index=None) # index=None means we don't get shifted table, unlike R
# convert to numpy
numpy_tab_xy = pandas_tab_xy.values; columns_tab_xy = pandas_tab_xy.columns

# read in a .csv as a pandas DataFrame if you have one
table = pd.read_csv('my_table.csv')
```

Essential libraries - 'The Scientific Stack'

You will almost always use these !

- **scikit-image**
  - Main easy-to-use image processing library with examples (common operations).
  - Slow, particularly 3D. Limited 3D image analysis.
  - Definitely not complete! Start here, Google/chatGPT for more specific, advanced uses
  - for Bioimaging Analysis, checkout this book. See bio-formats for more general handling of microscopy image formats

```
import skimage.io as skio
import skimage.transform as sktform
import pylab as plt
# read common image formats e.g. .png, .tif, .jpg. For .nd2 (install nd2), .czi (install czifile)
img = skio.imread('my_image.tif')
# save image
skio.imsave('my_saved_image.tif', img)
# resize image by linear interpolation
img_resize = sktform.resize(img, output_shape=(256,256), order=1, preserve_range=True)
# view image, figsize controls resoluton on-screen
plt.figure(figsize=(10,10)); plt.imshow(img_resize, cmap='gray'); plt.show()
```

Essential libraries - 'The Scientific Stack'

You will almost always use these !

- **scikit-learn**
  - Simplest library for machine learning. Every function has the same way of using.
  - Contains useful data preprocessing routines, and toy datasets
  - Primarily, a tour of classical machine learning techniques, which deliver very good performance and further improved with parameter tuning using `auto-sklearn` library

```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import pylab as plt

iris = datasets.load_iris() # load iris dataset, this has 3 features/dimensions
iris_sdscale = StandardScaler().fit_transform(iris.data) # standard scale each feature to be mean=0, std=1
pca_iris_sdscale = PCA(n_components=2).fit_transform(iris_sdscale) # use PCA to map to 2 dimensions to plot
# plot, and color points by the 3 iris types, and choose a colormap
plt.figure(figsize=(8,8))
plt.scatter(pca_iris_sdscale[:,0], pca_iris_sdscale[:,1], c=iris.target, colormap='Spectral')
plt.show()
```

Installing libraries

- All these libraries can be installed together using `conda` or `pip`.
  - `conda`:

    ```
    conda install numpy scipy matplotlib pandas scikit-image scikit-learn
    ```

  - `pip`:

    ```
    pip install numpy scipy matplotlib pandas scikit-image scikit-learn
    ```

- Beware:
  - Python packages may have complex and conflicting dependencies
    - e.g. PyQT (required by many GUIs)
  - Python packages may require compilation if not pre-compiled for OS
  - Best practice:
    - work in mainstream, stable Linux OS e.g. Ubuntu
    - create new conda environment for each project
    - install minimal set of libraries jointly upfront using e.g. `requirements.txt`

How to get started with a (new) library?

Look for examples and demos, follow-up with function documentation

1. Libraries typically all come with example scripts. Make sure to run to:
   - check library is correctly installed
   - check function usage
2. Some libraries have user guides and notebooks with detailed explanations, like a book
   - read them to understand topics - these are way more useful than a paper!
3. Check documentation of function for parameters, what format should the input be, what does it return
   - In `spyder`, using `ctrl-left click` automatically takes you to a function's source code
   - Access docstring with `help(function)` or online published function API references
4. Ask in relevant online general- and topic- specific forums:
   - StackOverflow: general computing
   - image.sc: scientific image analysis (biological mainly)
   - ChatGPT / Gemini AI and Google search engines (search online)
5. (Most important) Practice, read lots of code and documentation

Essential Skill: debugging code

There are only two types of error: 1) code syntax and 2) logic error

I am getting errors, it doesn't seem to work - how do we detect and fix? Here are some essential tools and tips.

- `print()`:
  - the simplest debugging tool, print your variables - are they what you expect?
- `assert()`:
  - do you have a variable that you know should be a particular value? Use `assert` which act like a brake, stopping the code when the condition is not met
- `help()`:
  - everything in Python is an 'object'. you can use help() to anything to get more information, even the variables you create or assign!
- **Plan your logic first:** what are the steps ? breakdown further to that you know how to do :
- **Construct and solve a toy problem:** check usage of a function or method

Activity

Use libraries to do exploratory analysis of patient dataset

(go to Jupyter notebook in the GitHub : activity_python-library-ecosystem.ipynb)

Data Visualization

Making publication-ready figures

Matplotlib

(the primary image plotting library)

- Matplotlib is a **beast** of a library and it is impossible to cover all functionalities
- Fortunately there is an extensive examples gallery, and cheatsheets

Basic Usage

- Figure with single plot

```
plt.figure() # create a figure canvas
plt.title() # give the figure a name
plt.imshow() # display a figure
plt.scatter() # plot some scattered points
plt.savefig() # save the figure (the extension specifies the image format)
plt.show() # display the figure
```

- Figure with multiple subplots

```
fig, ax = plt.subplots(nrows=2, ncols=3) # create a figure canvas, equally split 2 rows, 3 columns
ax[0,0].imshow() # display image in 1st row, 1st col
ax[0,1].plot() # plot line in 1st row, 2nd col
ax[1,1].bar() # plot bar graph in 2nd row, 1st col
```

Matplotlib

(the primary image plotting library)

Common plotting commands for science:

- line plot: `plt.plot`, (also used instead of scatter if marker is set)

```
plt.plot(x,y,'g.-',lw=3) # plot x,y as a line
```

- bar plot: `plt.bar` (plot vertical bars), `plt.hbar` (plot horizontal bars)

```
plt.bar(x,y,width=1) # plot vertical bar at x, width of 1
```

- error bar plot: `plt.errorbar`

```
plt.bar(x,y,xerr=x_errors, yerr=y_errors) # plot x,y as a line with error bars for each
```

- box plot: `plt.boxplot` and violin plot: `plt.violinplot`

```
plt.boxplot([[data_group_1],[data_group_2],[data_group_3]]) # boxplot of values from 3 groups
plt.violinplot([[data_group_1],[data_group_2],[data_group_3]]) # boxplot of values from 3 groups
```

- display 2D image: `plt.imshow` or `plt.matshow`

Matplotlib

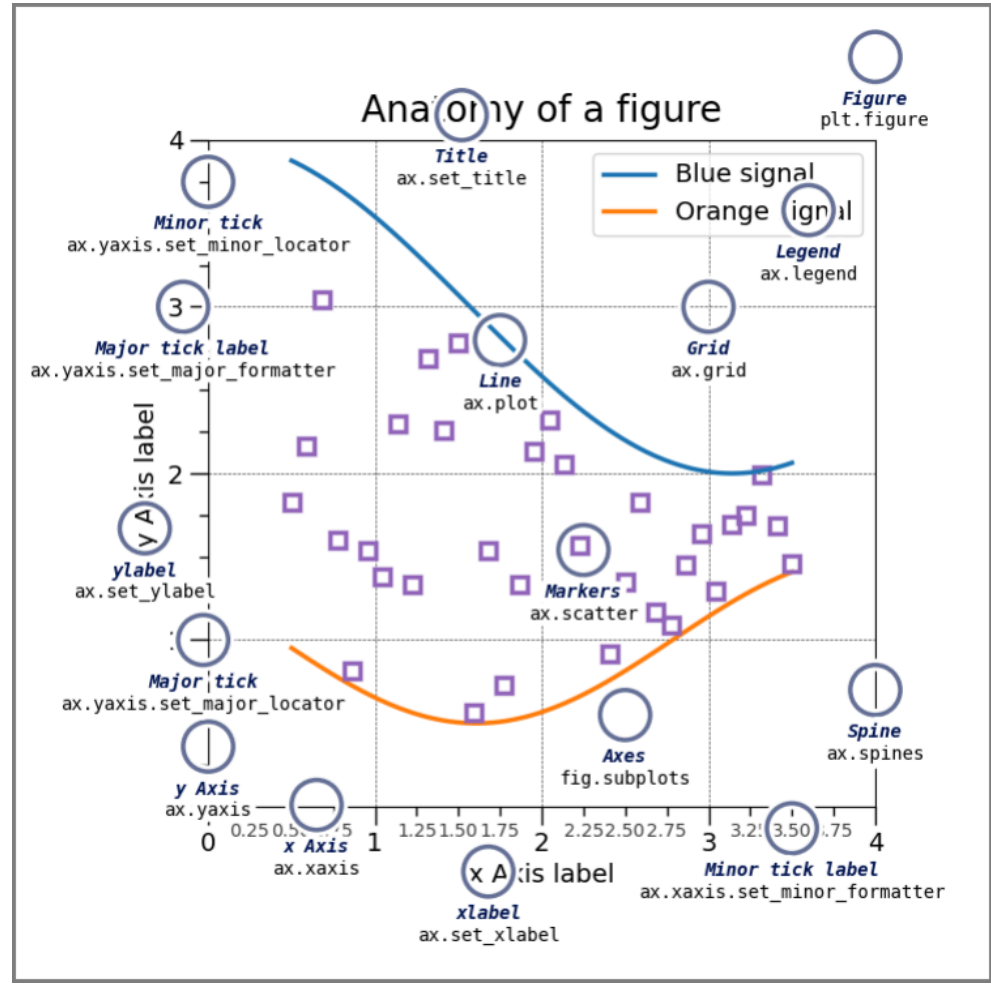
(Anatomy of a matplotlib figure)

- `fig = plt.figure()` : global plot canvas
- `ax = fig.add_axes([0.2, 0.17, 0.68, 0.7], aspect=1)` : plotting grid

Basic concepts:

- title: title
- spine: outer border
- grid: gridlines
- ticks: major/minor annotation of x-, y-axis
- label: name of x-, y- axis
- legend: plot legend

See: matplotlib figure anatomy



Matplotlib

Pros and Cons

- **Pros**:
  - Customizable - users have absolute control over every element
  - easy-to-install (i.e. always works, no complex dependencies)
  - many plot types with a function, working directly on numpy arrays
  - extensive developer support (longest developed, and actively developing)
  - extensive documentation and examples
  - export in many formats including vector graphics (.svg, .pdf)
- **Cons**:
  - needs tweaking of figure settings to look good
  - not all scientific plots supported or need lots of customizing (or do it in illustrator)
  - limited support for interactive and animated plots

Matplotlib Alternative: seaborn

Rapid prototyping of general scientific figures

- Install

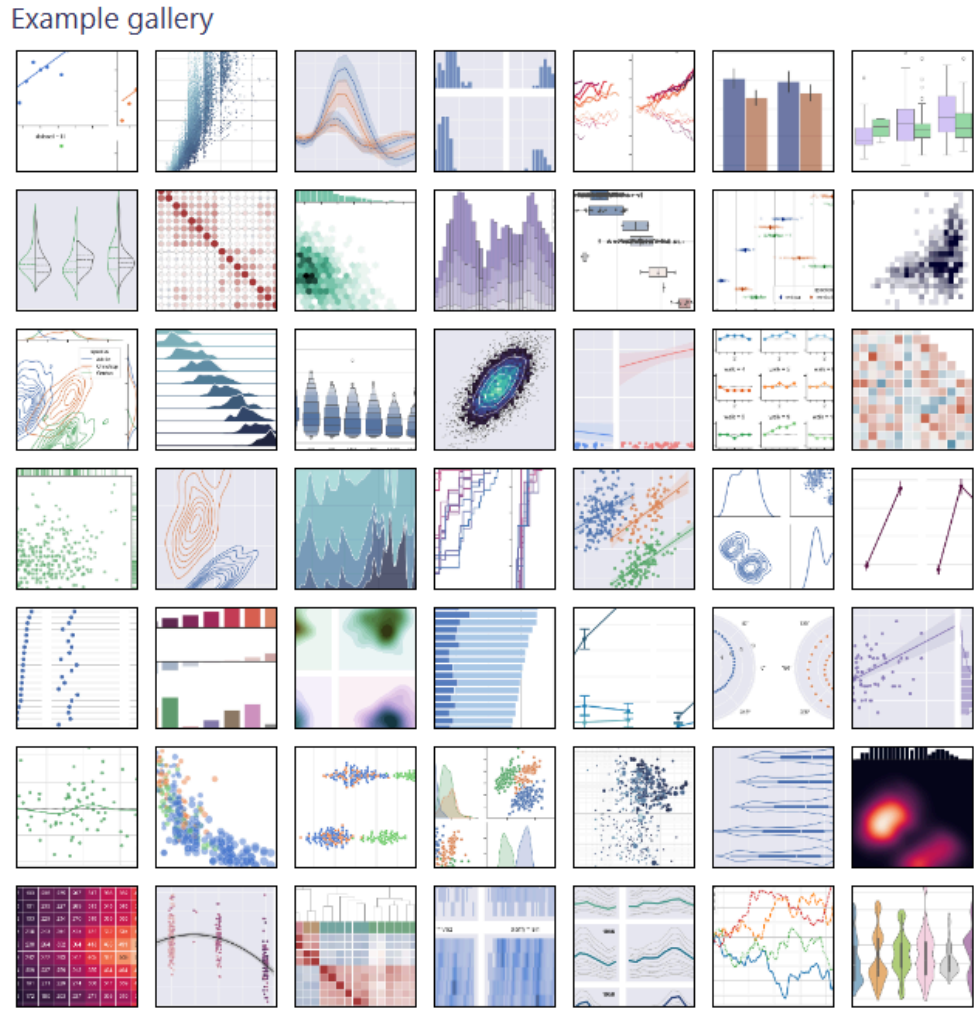
```
pip install seaborn
conda install seaborn
```

- Provide data as `pandas.DataFrame`
- Choose plot from gallery and reuse code

```
import seaborn as sns
violin = sns.violinplot(data=my_table, x='x_var_name
```

- Generate and use color palettes

```
color = sns.color_palette("husl", 8) # give me 8 col
sns.palplot(color) # visualize the palette
```

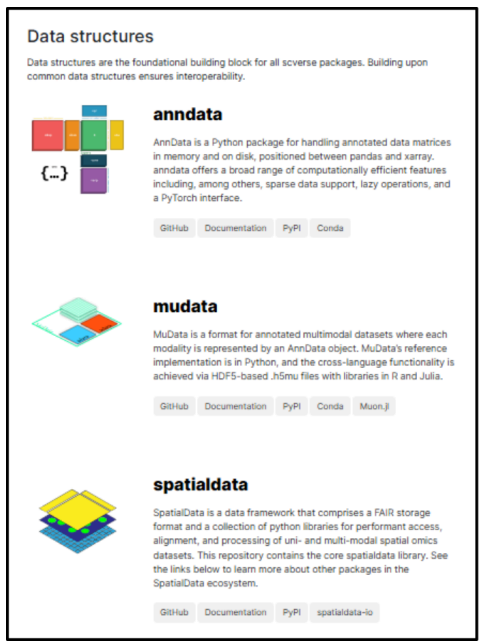


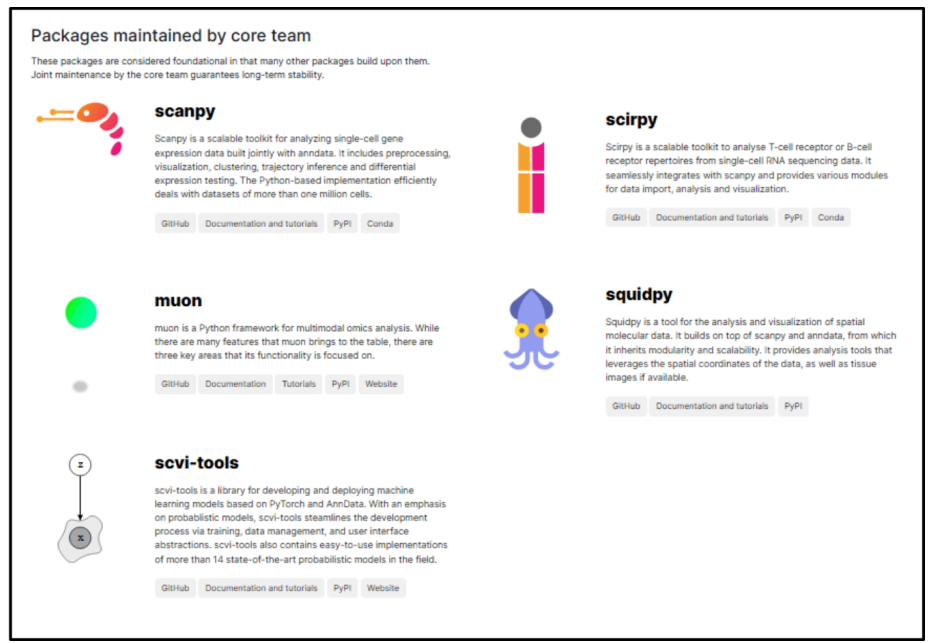
Matplotlib Alternative: anndata, scanpy and scverse

Scientific data structures and publication-ready scientific figures

- **scverse**:

Python ecosystem for single-cell transcriptomics and spatial transcriptomics analysis





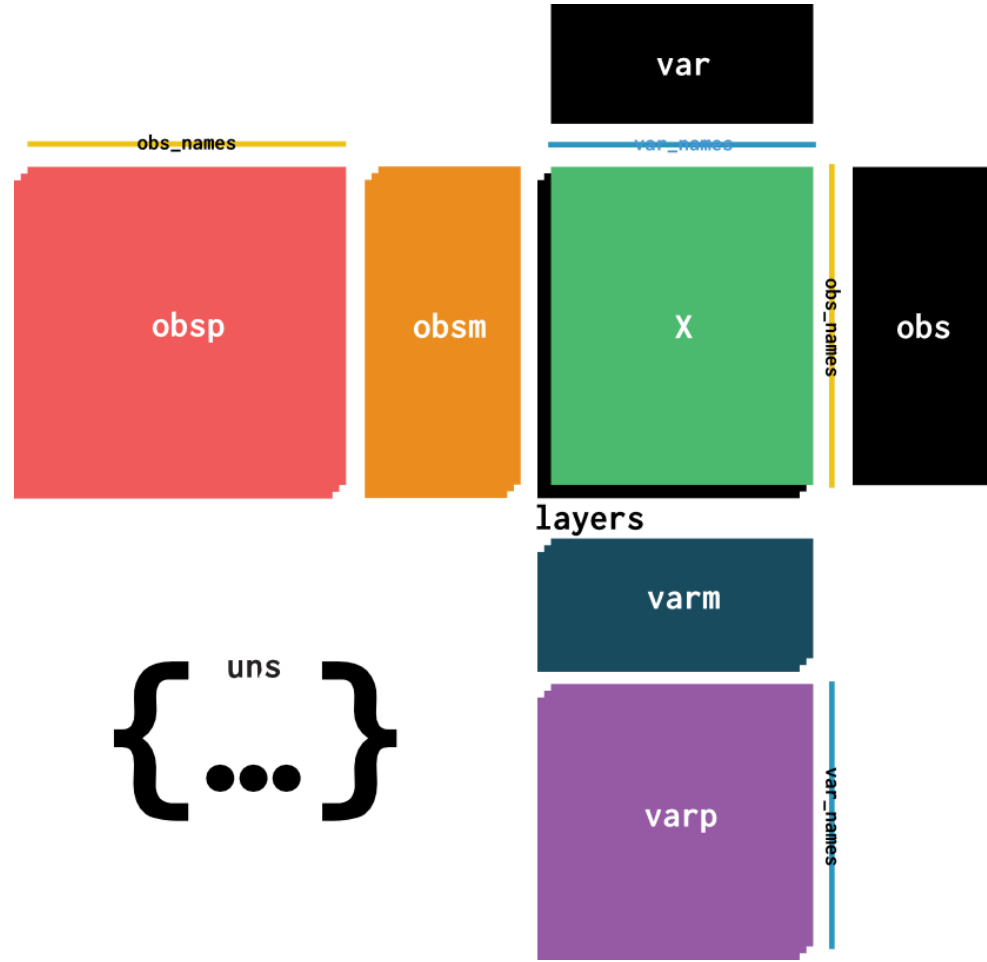
Matplotlib Alternative: anndata, scanpy and scverse

Scientific data structures and publication-ready scientific figures

- **anndata**: a data structure for science

```
pip install anndata
```

- an `anndata` object is a giant dictionary. This means you use 1 variable name for all your data, metadata and results (which are all dictionary items). This is all saved into a single `h5ad` file



Matplotlib Alternative: anndata, scanpy and scverse

Scientific data structures and publication-ready scientific figures

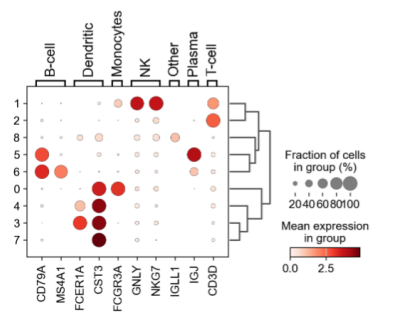
- **scanpy**: Python library for single cell transcriptomics analysis (all standard analyses)

```
pip install scanpy
```

- Publication-ready one line plotting of `anndata` object e.g.

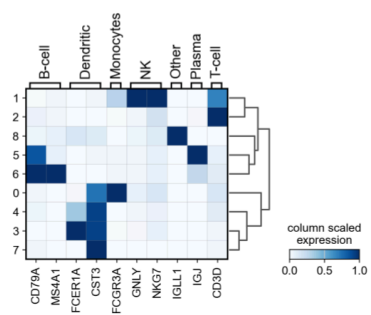
dotplot

```
sc.pl.dotplot(pbmc, marker_genes_dict, "clusters", dendrogram=True)
```



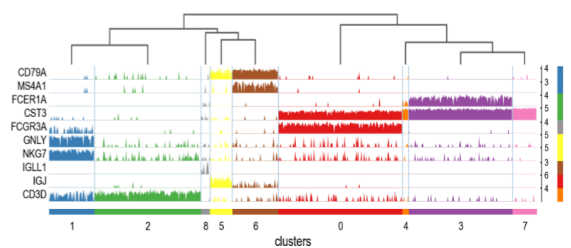
matrixplot

```
sc.pl.matrixplot(
    pbmc,
    marker_genes_dict,
    "clusters",
    dendrogram=True,
    cmap="Blues",
    standard_scale="var",
    colorbar_title="column scaled\nexpression",
)
```



tracksplot

```
ax = sc.pl.tracksplot(pbmc, marker_genes_dict, groupby="clusters", dendrogram=True)
```



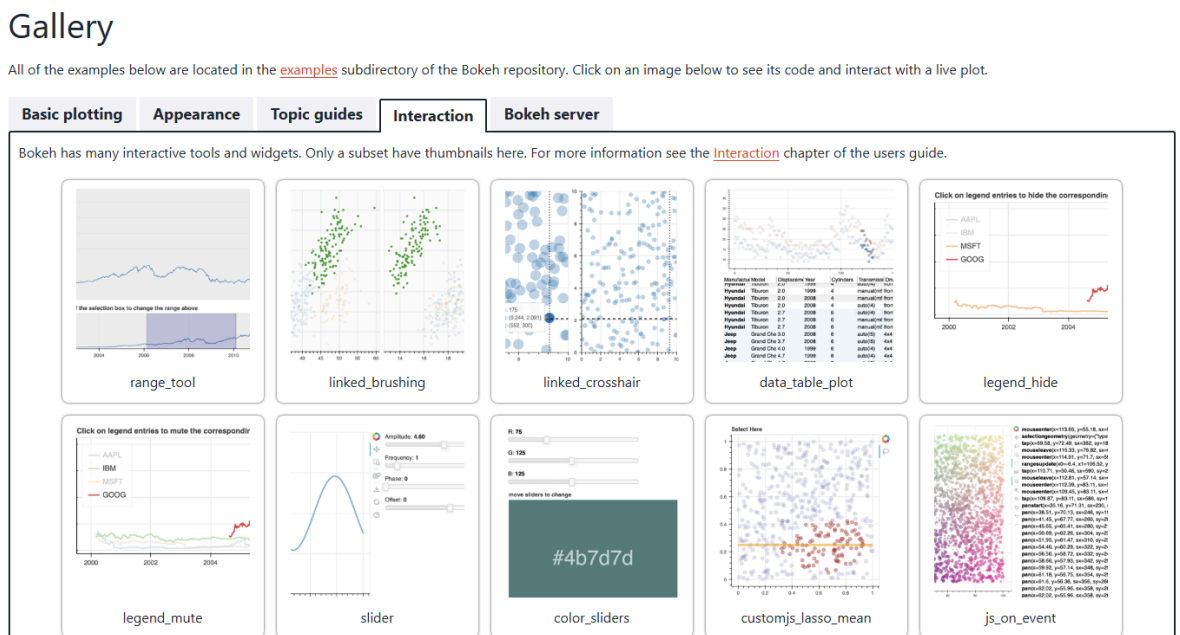
- Other plots tied to computational step e.g. UMAP, clustering, graphs

Interactive plotting in Python: `Bokeh`

Interactive plots in the browser that respond to user input

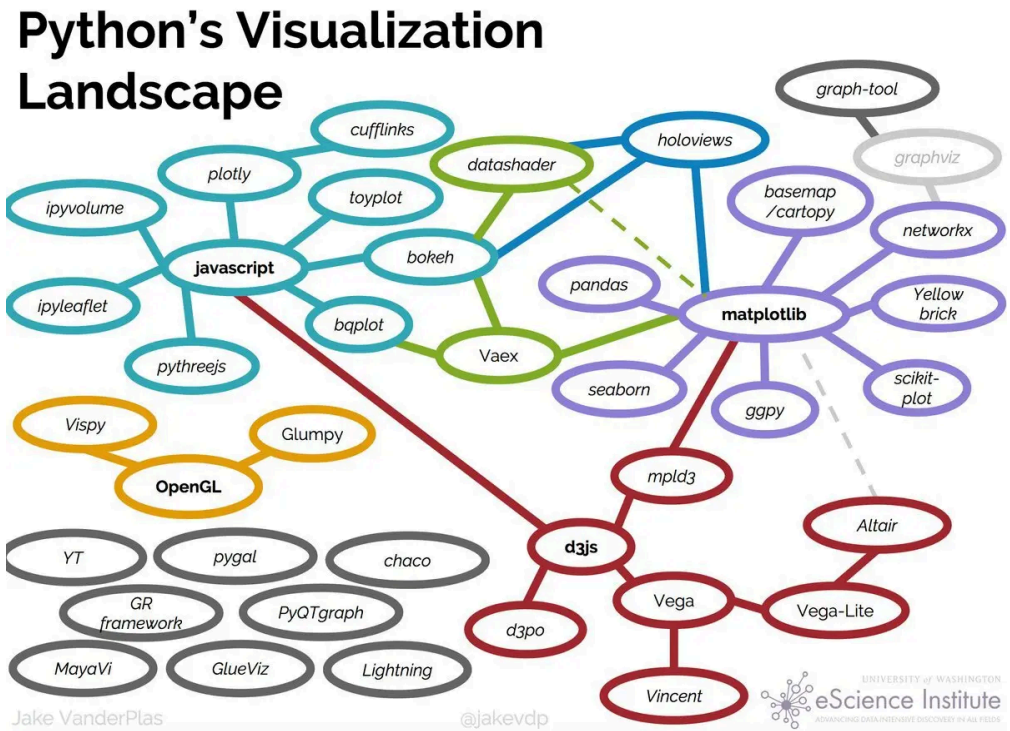
- `Bokeh` generates interactive plots in the web browser

```
pip install bokeh
```



Tip of the iceberg: countless possibilities

Many choices, once you have the data



Save your result, and you can use any you like!

How to save my data and results?

General saving options

- arbitrary python structures: `pickle`

```
import pickle

savedict={'save_1': my_dict, 'save_2':[list], 'save_3': numpy_array, 'save_4':'string'}

with open(savepicklefile, 'wb') as f:
    pickle.dump(savedict, f)
with open(savepicklefile, 'rb') as f:
  data = pickle.load(f)
```

- numpy arrays: `numpy.save` and `numpy.load`

```
with open('test.npy', 'wb') as f:
  np.save(f, np.array([1, 2]))
with open('test.npy', 'rb') as f:
  data = np.load(f)
```

Activity

Visualize properties of patient dataset

(go to Jupyter notebook in the GitHub : activity_python-data_visualization.ipynb)

Scientific Computing Basics

Leveraging powerful tools in Python for scientific data analysis.

Introducing `NumPy` (**Num**erical **Py**thon)

Effecient mathematical operations on N-dimensional data.

First, lets import the library:

```
import numpy as np
```

- `Lists` are versatile in that they can store heterogeneous data and can have "ragged" dimensions.
- Many applications, however, involve operations on rectangular N x M (or more) arrays of the same type, e.g. **matrix operations**. This is where NumPy really shines!

Fundemental type: **numpy.ndarray**

Create using any rectangular list (or tuple) of a single type:

```
arr = np.array([[1,2,3,4],[5,6,7,8]])
```

- Can also "initialize" with `np.zeros, np.ones, np.empty, np.random`, and more.

Introducing `NumPy` (**Num**erical **Py**thon)

- The power of `ndarray` is the ability to perform mathematical operations on them:

```
# scalar operations...
a = np.array([[1,2,3,4],[5,6,7,8]])
a = a * 2 + 1
print("a:", a)
# matrix operations...
b = np.ones(a.shape)
print("b:", b)
print("a + b:", a + b)
```

```
a: [[ 3  5  7  9]
[11 13 15 17]]
b: [[1. 1. 1. 1.]
[1. 1. 1. 1.]]
a + b: [[ 4.  6.  8. 10.]
[12. 14. 16. 18.]]
```

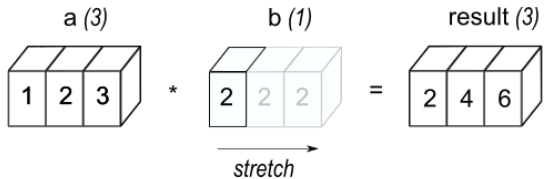
- In addition to *slicing* (recall w/ `list`), `ndarray` supports **Integer** and **Boolean** indexing:

```
# create some random integers
a = np.random.randint(-10,10, size=(3,3))
print(a)
# access only diagonals (Integer indexing)
print("diag:", a[[0,1,2],[0,1,2]])
# acess only > 0 (Boolean indexing)
print("positive:", a[a > 0])
```

```
[[-5 -6 -9]
[-7  6  7]
[ 9  0  2]]
diag: [-5  6  2]
positive: [6 7 9 2]
```

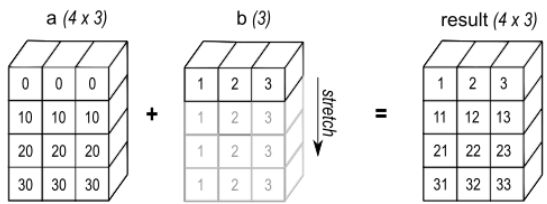
Broadcasting

NumPy handles operations between differently sized arrays by *stretching* along the larger axis.




```
a = np.array([1,2,3])
print(a * 2)
```

```
[2, 4, 6]
```




```
a = np.array([[0, 0, 0],[10,10,10],[20,20,20],[30,30,30]])
b = np.array([1,2,3])
print(a + b)
```

```
[[ 1  2  3], [11 12 13], [21 22 23], [31 32 33]]
```

And so much more!

Basic Math: `np.sin, np.cos, np.exp, np.log, ...`
Statistics: `np.mean, np.std, np.max, np.sum, ...`
Linear algebra: `np.transpose, np.invert, np.dot, np.cross, ...`
Fourier domain: `np.fft.fft/fft2, np.fft.ifft/ifft2, ...`

`NumPy` can do a ton! Let's not get too bogged down...
But as usual, read the docs.

Loading Images

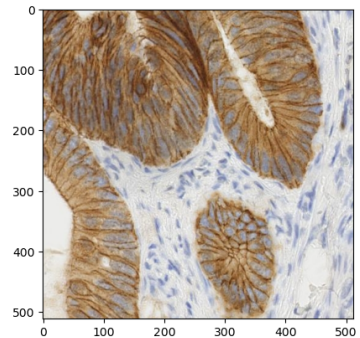
- Numerous packages exist which can load images easily. Some common ones are:

Matplotlib	`import matplotlib.pyplot as plt` `im = plt.imread('img.png')`
PIL	`from PIL import Image` `im = Image.open('img.png')`
OpenCV	`from cv2` `im = cv2.imread('img.png')`
Scikit-Image	`from skimage import io` `io.imread('img.png')`

Working with Images

- Let's load an image using some of the tools that we're already familiar with:

```
from skimage import io
import matplotlib.pyplot as plt

im = io.imread("..\\datasets\\images\\ihc.png")
plt.imshow(im)
```



- What are some of the properties of our image?

```
print(type(im))
print(im.shape)
```

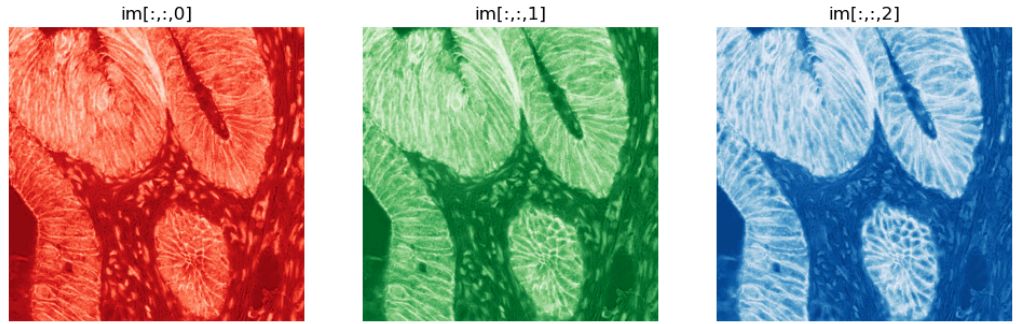
```
<class 'numpy.ndarray'>
(512, 512, 3)
```

- Many image analysis packages default to `numpy.ndarray` as the image datatype, making them convenient to work with.

Working with Images

RGB Color

- Note the 3rd dimension on our image: these are the **red, green, blue (RGB)** components of our image.



- If we `print(im)` directly...

```
array([[[156, 118,  81],
        [163, 125,  88],
        [156, 116,  81],
            ..., etc
```

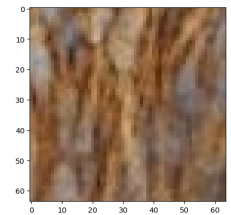
... we see that each pixel has the form **[r, g, b]**. Combined these values can form any color, but sometimes it may be useful to work with the RGB channels individually.

Manipulating Images with NumPy

The `ndarray` nature of images allows us manipulate them as with any NumPy array.

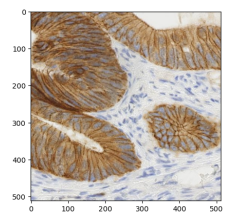
Cropping

```
plt.imshow(im[64:128, 64:128, :])
```



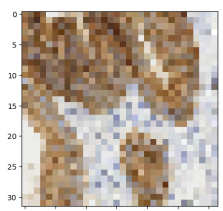
Rotate

```
plt.imshow(im.swapaxes(0,1))
```



Downsample

```
plt.imshow(im[::16, ::16, :])
```



Introduction to `scikit-image`

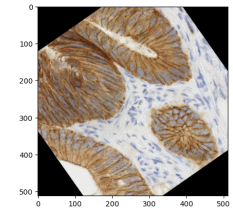
Designed for advanced image processing, building on `NumPy` to simplify and enhance image manipulation.

- **Advantages over Basic NumPy**:
  - Provides specialized image processing functions that would be complex to do with `NumPy` alone.
  - Handles various image formats, color spaces, and built-in filters efficiently.
  - Includes pre-built tools for common image tasks.
- **Key Image Processing Tasks**:
  - **Image Filtering**: Smoothing, sharpening, and edge detection.
  - **Segmentation**: Isolating regions of interest (e.g., objects, boundaries).
  - **Feature Extraction**: Identifying patterns and structures (e.g., corners, edges).
  - **Geometric Transformations**: Resizing, rotating, and warping images.
  - **Color Space Manipulation**: Converting between RGB, grayscale, HSV, etc.

Manipulating Images with `scikit-image`

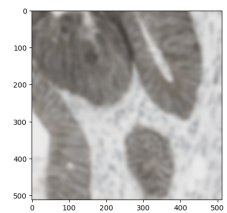
Arbitrary Rotation

```
from skimage import transform

plt.imshow(transform.rotate(im, angle=35.0, resize=False))
```



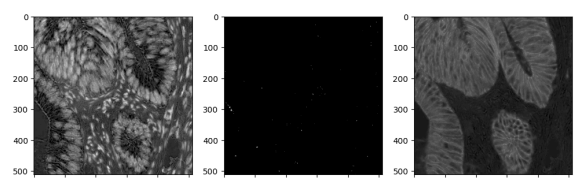
Gaussian Filtering

```
from skimage import filters

plt.imshow(filters.gaussian(im, sigma=5.0))
```



Spectral Unmixing and Contrast Adjust

```
from skimage import color, exposure
im_hed = color.rgb2hed(im)
fig, ax = plt.subplots(1,3, figsize=(12,4))
ax[0].imshow(exposure.adjust_gamma(im_hed[:,:,0], gamma=0.6), cmap='gray')
ax[1].imshow(exposure.adjust_gamma(im_hed[:,:,1], gamma=0.1), cmap='gray')
ax[2].imshow(exposure.adjust_gamma(im_hed[:,:,2], gamma=0.4), cmap='gray')
```



Introduction to **Sci**entific **Py**thon: `SciPy`

Python library built on `NumPy` that adds powerful tools for scientific and statistical analysis.

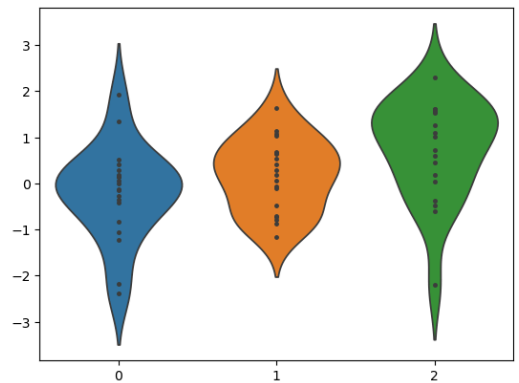
- **Advantages over NumPy**:
  - Provides advanced functionality for statistical analysis, hypothesis testing, and scientific computing.
  - Includes modules for linear algebra, optimization, signal processing, and integration, making it ideal for in-depth scientific analysis.
  - Offers high-level statistical tests (e.g., t-tests, ANOVA) and distributions, which are limited in `NumPy`.
- **Core Capabilities for Scientific Data**:
  - **Statistical Analysis**: Tools for probability distributions, descriptive statistics, and inferential tests.
  - **Hypothesis Testing**: Built-in functions for common tests, such as t-tests, chi-square tests, and ANOVA.
  - **Optimization and Fitting**: Functions for curve fitting, minimization, and root-finding.

Statistical Analysis with `SciPy`

T-Test for Independence

- Let's use what we have learned to generate some toy data and perform a T-Test for independence.

```
import numpy as np
# --------- Generate three random samples ---------
#                     sample    effect    variance
#                     size      size
v0 = np.random.normal(size=20, loc=0.0, scale=1.0)
v1 = np.random.normal(size=20, loc=0.1, scale=1.0)
v2 = np.random.normal(size=20, loc=1.0, scale=1.0)
# plot the data
import seaborn as sns
sns.violinplot(data=[v0, v1, v2], inner='point')
```



Always visualize your data!

```
import scipy.stats as spstats
result_01 = spstats.ttest_ind(v0, v1)
result_02 = spstats.ttest_ind(v0, v2)
print(f"[0<->1]: P={result_01.pvalue:.4f}, t={result_01.statistic:.4f}")
print(f"[0<->2]: P={result_02.pvalue:.4f}, t={result_02.statistic:.4f}")
```

T-Test Results:

```
[0<->1]: P=0.2008, t=-1.3019
[0<->2]: P=0.0065, t=-2.8815
```

Statistical Analysis with `SciPy`

One-way ANOVA

- *"Analysis of variance"* using the F-Statistic. Tests the null hypothesis that all means are equal.

```
from scipy.stats import f_oneway

# perform the one-way anova on our toy data
result = f_oneway(v0, v1, v2) # include all

print(f"ANOVA results:\nP = {result.pvalue:.4f}\nF = {result.statistic:.4f}")
```

```
ANOVA results:
P = 0.0104
F = 4.9470
```

- What happens to the ANOVA if we drastically reduce the "effect size (`loc`)" of `v2` from `1.0` to `0.2` ?

```
v2 = np.random.normal(size=20, loc=0.2, scale=1.0)

result = f_oneway(v0, v1, v2)

print(f"ANOVA results:\nP = {result.pvalue:.4f}\nF = {result.statistic:.4f}")
```

```
ANOVA results:
P = 0.3401
F = 1.0993
```

- `P > 0.05` : There is no significant difference between our datasets.

Activity

.\activities\activity_python-scientific-computing.ipynb

We will use the tools learned in this section to do:

- Image Processing
- Statistical Analysis