Introduction to Python

Foreword

`Python` is probably the most popular programming language by many measure these days, due to its simple learning experience and large ecosystem. In fact, I bet most of the people who are reading this note already know `python`. `python` has a long history and a huge community which you can probably do everything you want in `python`. However, being simple to learn also means it is easy to write bad code. Getting your calculation is one thing, building a nice package which people can use happily is another. There are many tricks and know-hows in `python` that you may not be aware of. And this is what we are going to focus in this lab.

Outline

Introduction to Python

Key Concepts

Python is an interpreted language

Everything is an object

Running a python script

`Python` files has the extension of `.py`. You can run a python script by running `python <script_name>.py`. For example, if you have a script named `hello.py`, you can run it by running `python hello.py`.

Introduction to Python

Basic Syntax

In this section, we are going to go through some basic syntax of python. We are only cover the minimum you need to know to write a simple insertion sort algorithm, especially most of you are already familiar with the `python` syntax.

Variables

To define a variable in python, you simply assign a value to a variable name. For example, to define a variable `a` with value `1`, you can do: `a = 1`. There are a couple of basic data types in `python`, like numbers, string, boolean. Three slightly more complicated datatypes are list, tuple, and dictionary.

List is a list of objects, which can be defined using the syntax `a = [0,1]`. A Tuple is an immutable list of objects, which can be defined using `a = (0,1)`. It is handy whenever you do not want things to change. A dictionary is basically a list but instead of accessing it by the index of the element, there are a list of key-value pairs, which you can access the values through their respective key. You can define a dictionary with the syntax `a = {"x": 1, "y": 2}` The dictionary in `python` is basically a hash table. They all can be accessed using `variable[index/key]`

One thing to remember is everything is an object in Python, meaning they (almost) all have some attributes and methods to themselves. If you have a background in `C` or some similiarly low level language, you may find be able to define something like `a = [0, "this", true]` blasphemous. There is certainly performance and stability implication to this feature, but I believe this flexibility is what makes `python` easy to get into.

Control flow

Functions

In order to define a function in python, you use the `def` keyword. For example, to define a function `add` which takes two arguments `a` and `b` and return the sum of `a` and `b`, you can do:

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

Now there is one tricky thing about `python`, which is the passed-by-object-reference. Some of you may have already ran into this in an unfortunate way, but for those who are not aware, here is a very sneaky failure mode one may spend hours trying figure out what's going on. Say I have defined a list `x = [0, 1]`, and I have a function that wants to modify the element in the list and return the modified list, what would you do? You may think you can do something like this:

```
def modify_list(x):
    x[0] = 1
    return x

x = [0, 1]
y = modify_list(x)
```

Now run try running this in a `python` interpreter, what do you get? Indeed, `y` will have the correct value. However, if you check `x`, you will find `x` is also modified. This is because when dealing with mutable objects such as lists or dictionaries, python passes a reference to the object instead of copying its value. This means when the interpreter is executing the line `x[0] = 1`, it is actually modifying the original object being passed it, causing this problem.

To avoid this, you can either make a copy of the object before modifying it, or you can return a new object instead of modifying the original object. For example, you can do:

```
def modify_list(x):
    x = x.copy()
    x[0] = 1
    return x
```

Copying the object may have some implication on memory usage and performance, so you may want to be careful when doing this.

Exercise: Writing an insertion sort algorithm

Now given the information above, let's try to write an insertion sort algorithm. The insertion sort algorithm is pretty simple. Imagine you are holding a hand of unsorted poker cards, and you want to sort the hand by the cards' rank. Let say we start from left and we are going to scan to the right. When ever we are about to sort a card, we compare the current card to the card on the left, if it has a lower rank than the card on its left, we swap their order (say I am trying to sort the second card which is a 2, and the first card is a 5, I will swap them). We keep doing this until we reach the end of the hand. At the end, the hand should be sorted.

Step 1 - Clone the repository

Fork this github template repository, then clone it to your local machine. You should see the file named "test_sort.py" in the root directory.

Step 2 - Implement the sorting algorithm

Open it with your favorite text editor, then change the body of the inerstion_sort function to an insertion sort algorithm.

Step 3 - Test the algorithm

Run the test_sort.py script with the following command:

```
python test_sort.py
```

If the test passes, you should see the following output:

```
Tests passed!
```

This means your sorting algorithm is correct.

Packaging code

Now the code is running, let's try to put it in a package so people has an easier time to use it. You can find more reference in the official python packaging guide. **PSA**: We are making the code base unnecessarily complicated for the sake of learning. The core code we have here is extremely small so there is absolutely no reason to go through all this hassle in practice, but this should be a good exercise to understand how to package a python project.

Step 1: Setting up virtual environment

Before we start playing with the code and package it, we do not want these potentially unstable code to be in our global python environment, which may cause some unexpected headache. So let's create a virtual environment for this project. You can do this by running the following command outside the project or in a directory where you want to store the virtual environment:

```
python -m venv insertion_sort
```

You should see a new directory named `insertion_sort` in the current directory. This is the virtual environment we just created, and it has some executable in it. To activate the virtual environment, you can run the following command:

```
source insertion_sort/bin/activate
```

You should see your terminal prompt has changed to something like `(insertion_sort) $`. This means you are now in the virtual environment. You can deactivate the virtual environment by running `deactivate`.

Step 2: Modules and imports

A `python` package usually contains many modules, which can be imported like `from package.module import function`. The goal here is structure our code such that it can be imported as a package.

Now the code is just living the root directory, which is not ideal when you start adding more and more files to the project. Instead, let's follow some standard structure and put our files in the right place.

In the root directory of your project, create a directory named `src`, within it create a directory named `insertion_sort`. Create a file named `sort.py` in the `insertion_sort` directory. Move the `insertion_sort` function from `test_sort.py` to `sort.py`.

There is one more step to make this work. In order to turn the `insertion_sort` directory into a module, you need to create a file named `__init__.py` in the `insertion_sort` directory. This file can be empty, but it is necessary to make the directory a module when it is packaged.

Now your project structure should look like this:

```
root
├── src
│   └── insertion_sort
│       └── __init__.py
│       └── sort.py
└── test_sort.py
```

Step 3: pyproject.toml

The next thing to add to the package is a description of project such that the standard build tool can package that information. I used to use `setup.cfg` to set up my project because that was the convention a couple years back. Now the standard way to set up a project is through `pyproject.toml`, which can still use `setuptools` as its backend. There are a lot of options to choose from for `pyproject.toml`, which we are not going to cover all of them. We are going to cover the minimum you need to know to build a binary. For a more complete tutorial and all the different options, you can find more information in the write your `pyproject.toml` guide.

At the root directory of your project, create a file named `pyproject.toml`. Add the following content to the file:

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "insertion_sort"
version = "0.0.1"
authors = [
  { name="Example Author", email="author@example.com" },
]
description = "A small example package"
readme = "README.md"
requires-python = ">=3.8"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.scripts]
insertion_sort_cli = "insertion_sort.sort:run_sort"
```

There are three parts in this file. The first part is the `build-system` section, which tells the build system what to use to build the project. The second part is the `project` section, which contains the metadata of the project, such as the name, version, authors, description, and so on. The third part is the `project.scripts` section, which tells the build system to create a binary named `insertion_sort_cli` that runs the `run_sort` function in the `insertion_sort.sort` module. Change the content in the `project` section if you want to.

We included the `script` section since I want to show you how to build a bbinary that also provide a command line tool. This is optional but it can be quite convinient for the user. That script is equivalent to running `from insertion_sort.sort import run_sort; run_sort()` in a python shell.

Step 4: Build and install

It is time to build and install our package! To build binary that you can distribute, first install `build` with `pip install build`, then you can run the following command:

```
python -m build
```

If everything goes well, you should see a new directory named `dist` in the root directory of your project. It should contain a `.tar.gz` file and a `.whl` file. These are the binary files that you can distribute to others, and usually it is uploaded to `pypi` with `twine` so people can install it with `pip`.

To install the package locally using the binary we just build, you can run the following command:

```
pip install dist/insertion_sort-0.0.1-py3-none-any.whl
```

Now you should be able to import the package in your python script. On top of that, you should also be able to run the `insertion_sort_cli` binary in your terminal. Try running it and see whether it works.

Building documentation

The structure of the code starting to look nice, however, nothing bring more trust to your code more than a nice documentation page. Once you have some serious documentation going, people will more likely to trust your code and use it. There are many different tools to build documentation in `python`. The more traditionally taught one is sphinx, but I personally find it unnecessarily complicated and looking dull, so I prefer to use `mkdocs` and style it with `material` instead. `Mkdocs` let you write your documenation pages in the `Markdown` format, which is pretty easy to write.

Step 1: Install mkdocs and build basic documentation

The first thing to do is to install `mkdocs`. With your environment activated, you can run the following command:

```
pip install mkdocs
```

Then in the root directory of your project, create a file named `mkdocs.yml`. Add the following content to the file:

```
site_name: Insertion Sort
nav:
  - Home: index.md
```

Now, create a fold named `docs` in the root directory of your project. In the `docs` directory, create a file named `index.md`. Add some random content to the file. Then in the root directory of your project, run the following command:

```
mkdocs serve
```

This will start a local server that serves the documentation. You can then click the link shown in the terminal and you should see your documentation page up and running. For more reference related to `mkdocs`, the official page is here.

Step 2: Style it with Material with mkdocs

Plain `mkdocs` actually look quite depressing too. Let's add some style to make it looks nice. I use Material for MkDocs to style my documentation page. You can install it by running the following command:

```
pip install mkdocs-material
```

Then in the `mkdocs.yml` file, change the content to the following:

```
site_name: Insertion Sort
theme:
  name: material
nav:
  - Home: index.md
```

Now if you go back to your doc page, you should see it suddenly looks much nicer already. There are many choices you can make to customize the look of the documentation page, you can find more information in the official documentation page.

Step 3: API documentation

Now this could be a bit more indepth since it needs to install your package, read the doc strings in your code and automatically generate the documentation page for API. I am not going into the detail of how to configure this, but you look for the voodoo magic I cooked up in one of my public repo flowMC. Here, I will show you the bare minimum of getting an API documentation page up and running. The package you will need here is `mkdocstrings` and its python handler. You can install it by running the following command:

```
pip install mkdocstrings mkdocstrings-python
```

In `docs` directory, create a file named `api.md`. Add the following content to the file:

```
# API Documentation

::: insertion_sort.sort
```

Then in the `mkdocs.yml` file, change the content to the following:

```
site_name: Insertion Sort
theme:
  name: material
nav:
  - Home: index.md
  - API: api.md
plugins:
  - mkdocstrings:
```

Now if you look at the page, you should see the API documentation there.

Writing tests

If you do not like writing documentation, there is a high likelihood that you do not like writing unit tests for your code. Unit tests are great way to catch any errors in your code, and when more contributors start to work on the same code, it is also a great way to catch any unexpected failure. The thing is, writing tests sound like some extra chores you have to do on top of the already dreadful development, so no body likes to do it. Here is how I trick myself into writing tests: we often write a bunch of scrappy scripts in development to se whether the code we created are function in the way we expected. I always create these scripts in either `test/integration` or `test/unit`, and only move them out if I want to make them an example. In this way, when I am done with my development, the tests are automatically there. The most popular package used for testing `python` code is `pytetst`, and it is quite easy to use. You can find more information in here.

Step 1: Install pytest

With your environment activated, you can run the following command to install `pytest`:

```
pip install pytest
```

Now for all the test we want to write, we have to write it with the prefix `test_`, so `pytest` know it is a test. This includes the test function name and the test file name.

Step 2: Move the test code to a test directory

I have this default test script `test_sort.py` in the root directory, let's create a `test` directory in the root directory and move the `test_sort.py` to the `test` directory. The next thing you will have to do is to take off the extra function definition in the test file, including the `__main__` block, and import the package we wrote.

Once that is done, you can run the following command in the root directory of your project:

```
pytest
```

This should run all the test in the `test` directory. If everything goes well, you should a long green line saying how many tests passed.

Rules of thumb

People have different opinions on writing tests. I usually keep two sets of tests in my test directory, `integration` and `unit` test. `integration` tests are aiming to test the entire code base in the way we anticipate the user to use it, while `unit` tests are testing the individual function in the code base. I usually write `unit` tests for all the functions I wrote, and I only write `integration` tests when I am done with the development and I want to make sure the code is working as expected.

When you write tests, you want to make sure they are as brutal as possible. The more brutal the test is, the more likely it is going to catch any unexpected failure. There is no point in writing a test that is going to pass no matter what. Use `assert <condition>` to make sure the test is going to fail if the condition is not met.

Introduction to Python

Best practices/Development tips

Virtual environment

In general it is good to have a virtual environment for each project you are working on. This is because different projects may have different dependencies, and you do not want to have a conflict between the dependencies of different projects. This is especially important when you are working on a project that has a lot of dependencies, or you are working on a project that has a lot of dependencies that are not compatible with each other. However, this means you may potentially create a lot of files and unwanted stress for the file system, so make sure you clean up the virtual environment when you are done with the project.

IPython

Typing

Linting and formatting

Debugging

Noteworthy libraries

Jax

I like to start with `jax` probably because I am a heavy `jax` user. Don't get me wrong, I use `PyTorch` too, but I use `jax` more because of its performance but also the workflow I have in `jax` is very close to the workflow I had before machine learning library became a thing. Basically, if you are familiar with `numpy` and `scipy`, you should find `jax` is basically `numpy` on steroids.

Flask

HoloViews

FastHTML

Too common/Too obscure

`numpy`, `scipy`, `pandas`, `matplotlib`, `seaborn`, `scikit-learn`, `pytorch`,