Life after SI 506

1.0 Coursera U-M Python courses (self-study)

Don't let the semester break go by without writing code. Consider signing up for a Python course on the Coursera program. As a University of Michigan student there is no charge for enrolling in U-M authored courses. Below is a select list of available offerings.

Coursera. Brenda Gunderson et al, Understanding and Visualizing Data with Python.

Beginner. Taught by LSA Statistics Dept faculty. Apply statistical concepts in Python using Numpy, Pandas, Statsmodels, Matplotlib, and Seaborn libraries. Course uses Coursera's Jupyter Notebook online environment.

Coursera. Paul Resnick et al, Python 3 Programming Specialization.

Beginner. Taught by UMSI faculty. Largely overlaps with SI 506 but includes several topics not covered in this course: list comprehensions, lambda expressions, and class inheritance.

Coursera. Steve Oney and Paul Resnick, Python Classes and Inheritance.

Intermediate. Taught by UMSI faculty. Overlaps SI 506 in certain respects but with a focus on class inheritance. This is course four in the Python 3 Programming Specialization.

Coursera. Chris Brooks, Introduction to Data Science in Python.

Intermediate. Taught by UMSI faculty. Learn how to work with the Pandas library in order clean, manipulate, analyze tabular data. This is course one in the five course Applied Data Science with Python Specialization.

Coursera. Charles Severance, Django for Everybody.

Intermediate. Taught by UMSI faculty. Learn how to build and deploy websites using the Django web framework, written in Python.

2.0 Intermediate Python

2.1 Python Styling (PEP 8)

Read PEP 8 Style Guide for Python Code (July 2001, Aug 2013). The guidelines emphasize *readability* and *consistency* "across the wide spectrum of Python code."

Jasmine Finer, "How to Write Beautiful Python Code With PEP 8" (RealPython, Dec 2018).

Jasmine summarizes ably the Python Community's Style Guide for Python Code (PEP 8).

💡 The Python Enhancement Proposal (PEP) is a technical design document for proposing new features and processes.

2.2 Multiple arguments: *args, **kwargs

A function can be designed to accept a variable number of positional arguments (`*args`) or a variable number of keyword arguments (`*kwargs`).

Lisa Tagliaferri, "How To Use *args and **kwargs in Python 3" (DigitalOcean, November 2017).

Davide Mastromatteo, Python args and kwargs: Demystified (Real Python, September 2019)

2.3 Classes

The Python `class` is a beautiful thing. When you define a class, say `Person`, you provide a blueprint, a template, or better yet, a model comprising attributes and methods that objects based on the class are provisioned with when created or instantiated.

Designing custom classes allows you to *instantiate* (i.e., create) multiple objects of the same type. This is no different than when you create a string, list, tuple, or a dictionary—you create an instance of `str`, `list`, `tuple`, or `dict`.

Python is an object-oriented programming language (OOP). Designing classes gets one thinking more intently about modeling an object's attributes and behaviors and the relationship that one object may have with another object.

In Python a `class` can *inherit* attributes and behaviours from a base or parent class. Leveraging class *inheritance* allows one to write specialized versions of a class in which an "*is-a*" relationship is defined between the parent class (the *supertype*) and the child class (the *subtype*) (e.g., a `Starship` is a `Vehicle`). Subtypes inherit supertype attributes and behavior. But a subtype is also free to both add additional attributes and methods as well as *override* supertype methods and attributes of the same name.

A `class` can also be created by combining objects. This is known as *composition*. A class instance (*composite*) can comprise one or more class instances (*component*). The relationship between a composite and its component is considered a "*has-a*" relationship (e.g., a `Person` has a `homeworld`—which could be an instance of a `Planet` class).

Eric Matthes, Python Crash Course, 2nd Edition (No Starch Press, 2019).

Read Chapter 9 "Classes".

David Amos, ["Object-Oriented Programming (OOP) in Python 3"](Object-Oriented Programming (OOP) in Python 3). (Real Python, Jul 2020).

Isaac Rodriguez, "Inheritance and Composition: A Python OOP Guide". (Real Python, n.d.).

Lisa Tagliaferri, "How To Construct Classes and Define Objects in Python 3" (Digital Ocean, March 2017).

Lisa Tagliaferri, "Understanding Class Inheritance in Python 3". (Digital Ocean, March 2018).

3.0 More advanced stuff

3.1 Type hints

Python 3.5 introduced support for type hints, a form of annotation useful during development and debugging. Python remains a dynamically-typed language but type hints help specify parameter types (e.g., `str`, `list`) for a function, method and/or variable are recoginized by linters such as `pylint` and other third party tools (e.g. Mypy) that use type hints to perform static analysis on your code and locate errors *before* they are encountered during runtime.

Basic syntax

```
variable: < type > = value
```

```
swapi_films: list = response['results']
```

```
def func(param: < type >, optional_param: < type > = default_value) ->
return_type:
```

```
def get_resource(url: str, params: Optional[dict] = None, timeout: int =
10) -> Union[dict, list]:
    """. . ."""
    ...
    return ...
```

Debanga Raj Neog, "Python Type Hinting: Beginner's Cheat Sheet" (Medium, May 2017).

Geir Arne Hjelle, "Python Type Checking (Guide)" (Real Python, n.d.).

Mypy, "Documentation".

3.2 Unit tests

The Python standard library includes a `unittest` module for writing unit tests (other test frameworks are also available). Learning how to write unit tests will improve not just the code you write but your development skills.

Python 3.x Official Documentation, "unittest — Unit testing framework" (python.org, nd).

Anthony Shaw, "Getting Started With Testing in Python" (Real Python, nd).

3.3 Decorators

Python functions are first class objects and can be passed an argument to another function. A function can also be nested inside another function. A *decorator* wraps a function and modifies its behaviour without

changing the function or its signature (e.g., parameters list). Decorated functions can be annotated using the "pie" `@some_decorator` syntax to indicate the function is "wrapped" by another function.

In the example below the function `get_swapi_resource()` is wrapped by the `functools.lru_cache` function. This results in the memoization of the function in which return values are cached based on the passed in arguments; calls to function that pass arguments that match previous calls use the cached value rather than the function's computed value.

```
@lru_cache(maxsize=64)
def get_resource(url: str, params: Optional[dict] = None, timeout: int =
10) -> Union[dict, list]:
    """..."""
    ...
    return ...
```

You can also decorate class methods using built-in decorators like `@classmethod`, `@staticmethod`, and `@property` as well as the class itself, such as `@dataclass` which provides a less verbose form of a class definition.

Geir Arne Hjelle, "Primer on Python Decorators" (Real Python, n.d.).

Geir Arne Hjelle, "Data Classes in Python 3.7+ (Guide)" (Real Python, n.d.).

3.4 LRU caching (memoization)

Cache the output of a Python function based on the arguments passed to it. This optimization technique is known as memoization and leverages the Python standard library's `functools.lru_cache`. A "memoized" function performs a computation and returns a value *only* once for each set of arguments passed to it. If the function is called with arguments passed to it previously, the return value is retrieved from the cache.

The example below illustrates the switch from dictionary cache implementation to the use of `functools.lru_cache()` which decorates `get_swapi_resource()` (see above).

functools.lru_cache()

See `lru_cache()` decorated `get_resource()` above.

```
# List of Film objects
for film in films:
    for i, url in enumerate(film.planets):
        planet_data: dict = utl.get_resource(url) # response cached
        planet_data = utl.convert_str_to_int(response)

        planet = Planet(
            planet_data['url'],
            planet_data['name'],
```

```
            planet_data['diameter'],
            planet_data['population']
            )

        film.planets[i] = planet # replace URL str with film object
```

Dan Bader, "Memoization in Python: How to Cache Function Results" [https://dbader.org/blog/python-memoization] (danbader.org, n.d.).

Santiago Valdarrama, "Caching in Python Using the LRU Cache Strategy" (Real Python, November 2020).

3.5 Logging events

The Python standard library includes a logging module for recording runtime events and streaming them to the screen and/or writing them out to a file. Logging is useful both for debugging and tracking program actions.

Abhinav Ajitsaria, "Logging in Python" (Real Python, n.d.).

Akshar Raaj, "Understanding the Python Logging Library" (Medium, November 2020).

3.6 Regular Expressions

Defining search patterns that can match complex character sequences requires the use of *regular expressions* (a.k.a regex or regexp) and the Python `re` module.

Thomas Nield, "An Introduction to Regular Expressions" (O'Reilly Media, Inc., June 2019).

💡 For RegEx testing/debugging I recommend the online tool regular expressions 101.

3.7 Pandas, Numpy, and Matplotlib

If you are interested in data, data manipulation, and data science, then start learning Pandas now. Pandas is built on top of Numpy, the base package for scientific computing. You will learn how to work with series, data frames stored as numpy n-dimensional arrays and matrices. Then learn how to visualize your data starting with the Matplotlib package. You can also do your work and publish your results using a web-based Jupyter notebook.

George McIntire, Brendan Martin, and Lauren Washington, "Python Pandas Tutorial": A Complete Introduction for Beginners".

3.8 Generators

A Generator is a high-performance function that simplifies the building of iterators. It returns what is known as a "lazy iterator" object, using a `yield` statement to suspend execution of the loop temporarily in order to pass a value back to the caller "on demand" without actually exiting the function. When the function is next called loop iteration continues yielding back the next value in the sequence. Generators come in handy when working with very large data sets (e.g., large CSV file).

Dan Bader, "What Are Python Generators?" (dbader.org, nd).

Dan Bader, "Generator Expressions in Python: An Introduction" (dbader.org, nd).

Kyle Stratis, "How to Use Generators and yield in Python" (Real Python, September, 2019).

4.0 Enhancing your dev environment

4.1 Python virtual environments

When working on unrelated Python projects, I create a "virtual environment" for each. Doing so creates an isolated development environment that allows each project to define its own dependencies independent of other projects. This includes not only package dependencies and their versions but also the Python version required for the project.

Real Python, "Python Virtual Environments: A Primer" (Real Python, nd).

Anthony Whyte, "Python virtual environments" (si506.org, nd)

4.2 Git and Github

Git is a distributed version control system for tracking changes in source code and other files. Github is a hosting platform for developers (and others) who version their work using git. Both the system and the platform are designed to support collaborative work among programmers, but versioning your solo work and maintaining copies in the cloud is smart practice.

Roger Dudler, git - the simple guide.

I love this guide.

Ross Conyers, "Learn Git in 3 Hours".

Video format. Videos grouped into four chapters. Chapter 4 focuses on Github.

- Chapter 1: Version Control and the Terminal
- Chapter 2: Learning the Basics of Git
- Chapter 3: Branches and Workflow
- Chapter 4: Advanced Git Workflow

4.3 Homebrew and Choco

Both macOS and Windows users can benefit from `pip`-like package managers that handle software installs.

For my Mac I use Homebrew, a macOS package manager, to acquire and maintain many of the software packages that I use on a daily basis, including Python. The Homebrew approach is but one way to manage software installs. In the case of Python on a Mac it is often described as the recommended way to install and maintain it.

For Windows I turn to Chocolatey to manage many of my installs.

5.0 Web development

If you are interested in web development consider exploring Django, Flask, and/or FastAPI. These Python web frameworks are designed for rapid development of database-driven web applications.

5.1 Learn Django

Charles Severance, Django for Everybody (DJ4E).

Django Girls, Django Girls Tutorial.

Mozilla, MDN web docs. Django Web Framework (Python).

Vitor Freitas, "A Complete Beginner's Guide to Django".

5.2 Learn Flask

Miguel Grinberg, Flask Web Development, 2nd Edition (O'Reilly Media, Inc., 2018).

Miguel Grinberg, "The Flask Mega-Tutorial".

An impressive series of blog posts that get you up an running with Flask.

Miquel Grinberg, "The New and Improved Flask Mega-Tutorial".

What started as a series of blog posts is now a formal, fee-based, course offering on Flask web development.

5.2 Learn FastAPI

FastAPI, "Tutorial - User Guide" (FastAPI, nd)

FastAPI, "FASTAPITUTORIAL" (FastAPI, nd)

Ben Gorman "Building A Simple CRUD Application With FastAPI" (GormAnalysis, nd)

Sebastián Ramírez, "Using FastAPI to Build Python Web APIs" (Real Python, nd)

6.0 Database design and development

If you are new to database design and development consider starting with SQLite, an in-memory database that is easy to install and maintain. Additionally, you need to learn the Structured Query Language (SQL). You start learning both by taking Dr Chuck's "Using Databases with Python" Coursera course.

Coursera, Charles Severance,"Using Databases with Python".

Intermediate. Taught by UMSI faculty. Learn how to build and deploy websites using the Django web framework, written in Python. This is course four in the five course "Python for Everybody Specialization".