Getting Started With Python

Autodesk University 2019
SD322308-L
Murano 3205, Level 3, Lab 6
Tuesday, November 19 2:45pm

Gui Talarico
@gtalarico

Learning Objects

- 🙉 Learn how to set up a programming environment for Python
- 🖥️ Learn how Python code is interpreted and executed
- 📝 Learn how to read and write files using Python
- 📘 *Learn the basic data types and flow-control features of Python*

Description

This is an introductory class to Python—no previous experience required. The class will walk through basic concepts of Python and how to set up a Python programming environment. It will cover Python's basic data types, flow control mechanisms, and some of the native libraries that you can use to develop your own automation tools in Python. At the end of the session, attendees will be able to develop a working Python script that can read and write files that can be used to transfer data between different platforms.

Speaker

Gui Talarico is a Software Engineer at Airbnb's Samara Design Studio. Prior to joining Airbnb, he was a Software Engineer at WeWork where his work focused on the integration of software development, physical space data, and design processes.

Beyond his professional work, Gui is an active contributor to open source projects, and online communities, including is Revit API Docs, an online Documentation platform for the Revit API. Since launching in late 2016, RevitApiDocs.com has received over 5 million views and has been widely recognized as a valuable resource to the Revit developer Community.

At Autodesk University 2017, Gui Talarico received the Best New Speaker Award as well as a Top Rated Speaker Honorable mention for his Hands on Lab session Untangling Python.

Links

- Presentation
- Github Repo

Assistants

- Timon Hazell
- Pablo Gancharov
- ?

Software Requirements

- [ ] VS Code
- [ ] Python37 → We will install together

Python Overview

What is Python?

Python is a high-level, general-purpose, open-source programming language. The language is widely used in a variety of fields, including mathematics, data-science, and web-development.

Python is also an interpreted language, meaning, its code is *interpreted* and does not require compilation. The program that interprets and executes python source code is called an Interpreter.

The most popular Python Interpreter implementation it's CPython, which is written in the language "C". CPython is the most popular, or *reference* Python implementation, and will be the implementation referenced in this session

If all this sounds confusing, do not worry. Understanding these concepts is not required to to learn Python. But if you are curious and want to dig deeper, these wikipedia entries explain these concepts in greater details:

* Programing Paradigms
* Python
* Interpreter

What is IronPython?

IronPython is an implementation of the Python specs created written in C#. Fun fact, "Iron" stands for "*Implementation Runs On .NET*", according to one its maintainers, Alex Earl (Talk Python to Me, Episode #74 -Past, Present, and Future of IronPython @ 9:20).

The C# implementation allows the interpreted python code to interact directly with other applications compatible with Microsoft's .NET framework (C#, F#, VB, etc) through the Common Language Runtime (`clr` module).

This language interoperability has made IronPython a popular language for creating embedded scripting environments within Windows applications that use the .NET framework. Some applications that offer embedded python scripting environments are Rhino + Grasshopper, Autodesk Maya, and of course, Dynamo.

While Python code can be written to be compatible with both mainstream Python and IronPython, this is only true when the code doesn't take advantage of implementation-specific features. For example, code that requires .NET interoperability requires the ironpython-exclusive `clr` module. Conversely, Python code that depends on libraries that rely on native C code cannot run in IronPython (Numpy, Pandas, etc).

Another disadvantage of IronPython is that development is several years behind CPython. The latest stable release is IronPython 2.7, while CPython is 3.8. Python 2.7 will reach end-of-life in 2020 and will no longer supported by the python community.

We will not be using IronPython in this session. The topic is covered here is because IronPython is used widely in AEC industry so it's helpful to understand the differences and when one might chose one over the other.

CPython x IronPython: Which One Should You Use?

If you are using Python within Revit or Rhino, you *generally** need to use IronPython. If you are using Python to develop scripts to be used outside of these application, you are better of targeting CPython.

This class will focus on using CPython, and in the context of all this document, CPython and "Python" will be used interchangeably.

If you are interested in using Python within Dynamo, my AU 2017 session on Python for Dynamo is a better resource - https://github.com/gtalarico/au2017

* There are is experimental project called PythonNet which allows CPython to integrate with .NET's clr. It has some limitations, it does work as advertise and has shown substantial improvement over the last few years

Why Learn Python

Recognized as having a shallow learning curve, Python has become a popular choice for education environments and beginners. The syntax and language-constructs are simple and promote code-readability - when well written; its code is often considered easy to read even by non-programmers.

Python's language design also makes it an excellent language to create short macro-like scripts and automation logic due to its conciseness and comprehensive standard library (modules that are built-in and shipped with the language).

Adding a language like Python to your toolset enables you to develop tools and workflows that are often faster than *pure* visual-programming, but also allow you to overcome limitations where the programs fall short.

As you develop your Python skills by writing simple scripts, you can start expand your usage of Python to develop websites, Web scrappers and APIs, machine learning models, and much more.

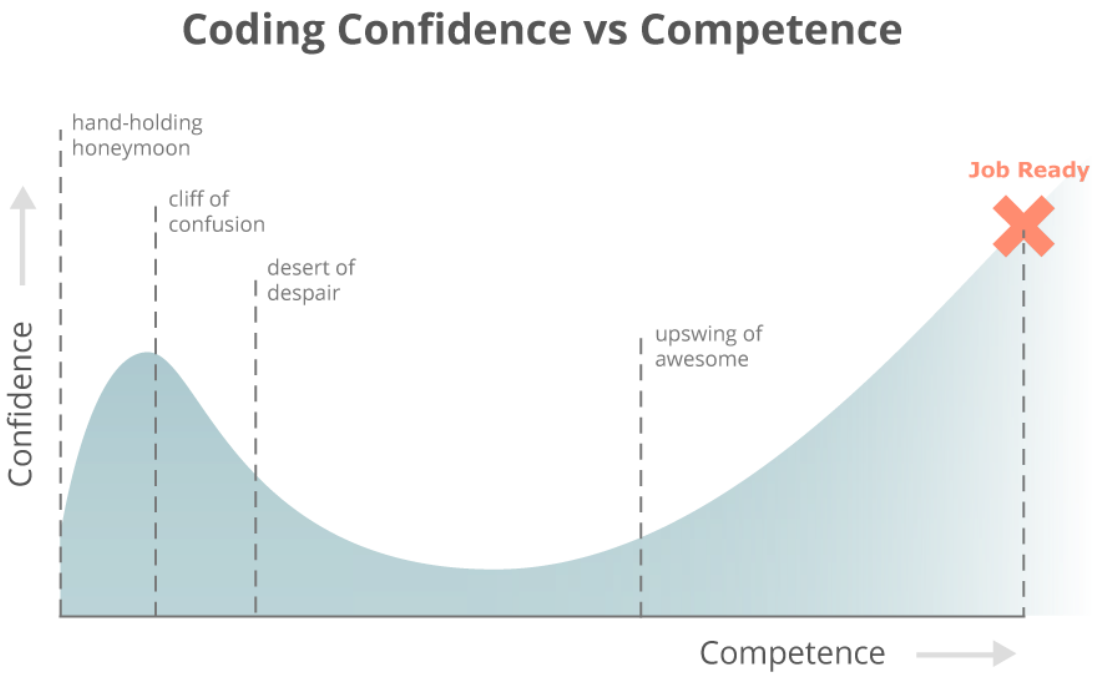
Learning Journey

Learning a new language is no easy feat. It takes time and a lot of patience. Each person will have their own unique journey and destination. Some may want to simply automate some work tasks, while others want change careers and become a software developer.

If Python is not your first language, you are in luck!-You can expect your learning curve to be substantially shallower!

As you start your learning journey, you might also enjoy reading this article by Erik Trautman

Why Learning to Code is So Damn Hard. The article offer interesting analogies and insights into the challenges we go through when learning a new programming language.



https://www.thinkful.com/blog/why-learning-to-code-is-so-damn-hard/

First Things First - Learning the Syntax

Computer languages have a clearly defined syntax which defines how the language is structure, which words have special meaning, and how operations like variable assignment and loops are defined.

Learning the syntax of a language is not hard. Most people can spend one or two weekends going through a python tutorial and will have learned how to write most language constructs.

Most beginners manage to learn the syntax, but then struggle with what comes next:

- How to troubleshoot and debug your code
- How to combine different language constructs together into useful software
- How to distribute or run your code on different contexts (eg. a server)

These skills take more time, and one can spend their entire career continuously refining them. In the next section we will define a framework and suggestions to help you increase your learning speed and effectiveness.

Language Concepts and Idioms

As a beginners, we do not want to reinvent the wheel. Instead of trying to create our own pattern from scratch. It's much easier to follow well community conventions and to reuse establish patterns.

There multiple ways to do achieve the same result with code, although often some are considered "better" than others. Python code that follows accepted language conventions and express intent clearly is often referred to as more "pythonic".

Below are two relevant PEPs (Python Enhancement Proposal) that are considered important documents in establishing the language's guiding principles:

Pep 20 - Zen of Python

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> ...
> There should be one-- and preferably only one --obvious way to do it.
> ...

Pep 8 - Style Guide for Python Code

Pep8 defines standards for code-formatting and naming conventions. While you should not attempt to memorize in it's entirety, it is helpful to become familiar with it and revisit it as needed.

I also highly recommend the use of linters. Linters are programs that analyze code and return a list of warnings or errors. Visual Studio Code, which will be used in this session, supports several linters out of the box.

Idiomatic Python

Written by Jeff Knupp, this book documents the good and bad ways of writing python. It's short and simple but fun to read and I highly recommend. You can get a sneak peak of the content on this Blog Post. Learning resources like this are always a great way to immerse yourself into the language, practice reading source code, learn better patterns.

Application Patterns

Complex Application patterns are beyond the scope of this session. For beginners, the best way to improve your ability to develop clean and maintainable code is write code, iterate, and read application source code.

A good place to read application source code is the Awesome Python Application repository.

Troubleshooting and Debugging

Troubleshooting is the process of identifying errors in your application (eg. why it doesn't run, or why is a certain error being raised). We will learn more about identifying errors as we start to code, but it's helpful to familiarize yourself with the built-in Python Errors & Exceptions.

Debugging, is a form of troubleshooting, but usually refers to the process of identifying a bug in your application or understanding why it's not behaving the way you expected.

There are two primary ways to debug our program: print it's output, or insert a breakpoint and inspect the values interactively. While the "print" method is easiest and most common, using breakpoints is more powerful and often more effective. Later into the session will will explore both methods.

Getting Help

Learning how to find solutions online or how to ask for help is one of the most valuable skills a beginner can have. If you run into an error, often a simple search will help you find the solution. However, if the issue you have is specific to your application, or if you are not sure how to build something, being able to ask a good question is key.

A well asked question is more likely to a get a fast and good response, while a poorly structured question will likely be ignored or even removed from Q&A platforms.

This post on Stackoverflow outlines some important pointers to consider: https://stackoverflow.com/help/how-to-ask

If your issue is related to an open source library, the issues section on it's repository is also a great place to search for solutions.

When asking others for help, be sure to consider the following:

- Are you giving enough context?
- Are you providing a way for someone to reproduce?
- Is your question clear and property formatted?
- Have you shown effort and included how you have attempted to solve the issue?
- Always be kind and respectful

Other Learning Resources

Documentation

Python official docs can be found at docs.python.org.

I highly recommend all beginners become familiar with it's structure and content.

In particular, I recommend you read at least the Tutorial and the Built In Functions sections.

Dive into Python Book

http://www.diveintopython.net/

Tutorials

One you are familiar with the basic syntax, tutorials are an excellent next step to learn how language constructs are assembled to together to achieve a purpose.

- A list of simple automation tutorials
  automatetheboringstuff.com

- Mathematical and computer programming challenges
  projecteuler.net/archives

- Create a website using Django
  docs.djangoproject.com/en/2.2/intro/tutorial01/

- Crete a website using Flask
  blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

- Solve some problems on Leetcode
  https://leetcode.com/problemset/all/?difficulty=Easy

Community and Language Immersion

- Read Python Source Code on Github regularly
- Read Python Questions on Stackoverflow regularly
- Learn How to Ask and Answer Questions on Stackoverflow
- Listen to Talk Python to Me and Python Bytes
- Attend your local Python Meetup

Learning Checklist

If you are not sure where to start, you can use the list below:

- [ ] Read two sections 1-4 of Python Tutorial
- [ ] Complete One project from automatetheboringstuff.com
- [ ] Reach 100 Reputation Points on Stackoverflow by asking and responding to questions
- [ ] Web Track: Start the Flask Mega Tutorial

Setting up a Python Developing Environment

Editors

Before we get into our Python installation and environment, we will first talk about editors.

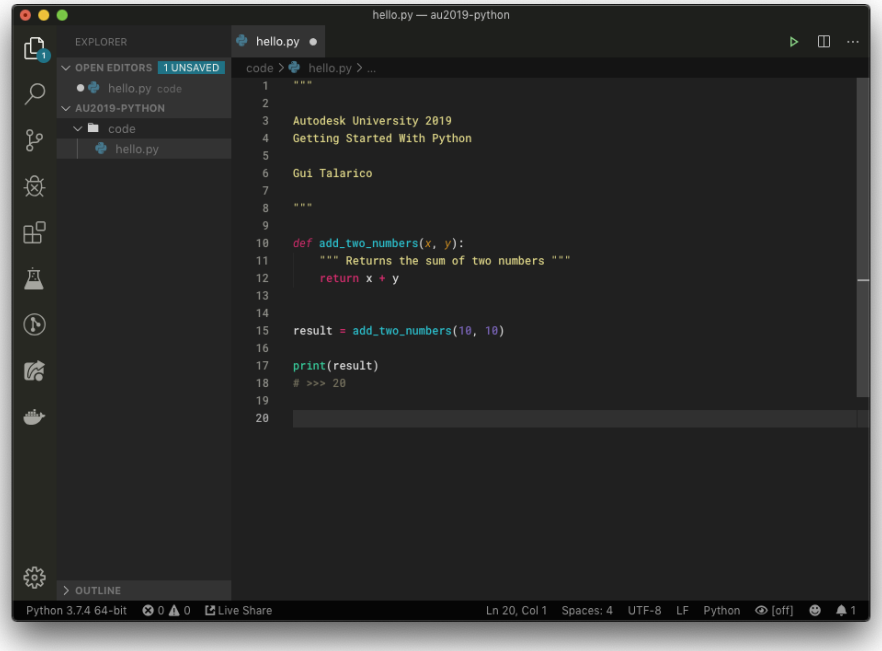
Python code can be written using any text editor, although most developers use robust text editors like Atom, Sublime, Visual Studio Code, or Notepad ++.

Integrated Development Environments like PyCharm or Visual Studio can also be used, however, I find they are often intimidating for beginners.

You can also write your code directly in embedded editors such as the one provided by Dynamo, although they usually lack many of the features provided by the alternatives above like code completion, code linting, and file management.

You should use whichever editor feels more intuitive to you. This class will teach use VS Code editor and execute code directly through the Python Interpreter.

Head over to code.vistualstudio.com and install a copy on your computer.



Visual Studio Code Terminal

Unless you are already comfortable using the a terminal/command prompt, the builtin Terminal embeded in Visual Studio Codeeditor can remove complexity beginners have to deal with.

This will make it easier for us to write and edit code in one environment.

Launching a Terminal

To launch a new terminal go to the **Terminal** and click **New Terminal.**

Python Installation

Learning how to properly install and setup an intuitive development environment can a difficult hurdle to overcome. Understanding how the language works and how your code executed can give beginners a much needed confidence to progress quickly

While installation techniques vary between Operating systems and version requirements, we will focus on installing Python for Windows.

- Head of to python.org/downloads/ and download version **3.7.5**
- If you are not sure which installer to get use *Windows x86-64 executable installer*

Why **3.7.5** ? Python3.8.x was released very recently (Oct 2019) and you more likely to encounter library compatibility issues. 3.7.5 is the latest stable version of the 3.7.x releases

Installation Steps

If you comfortable following the Installation wizard on your own, go for it. If you need additional help, you can refer to the step-by-step instructions on Real Python Installing Python Tutorial.

However you decide to setup your installation, just make sure you check the box "Add Python 3.x to PATH". This will ensure the Python interpreter can be found when you type `python` on the command prompt

Checking your Installation

Open a Command Prompt and enter `python`. If you installed it correctly, it should start a python interpreter

```
C:\>python

Python 3.7.3 (default, Jun  4 2019, 11:29:12)
[Clang 11.0.0 (clang-1100.0.20.17)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Writing and Executing Python Code

Using a Command Prompt

Learning how to use a command prompt, or terminal, is an important skill to learn as most programing languages require its use. While no advanced knowledge is required, we will need to learn at least a few commands to find our way around the prompt:

```
C:\>dir
# list files and directory

C:\>mkdir test
# makes a folder named test

C:\>cd test
# changes current directory to test

C:\test> explorer .
# open a windows explorer window at current directory
```

Where to Save Your Files

Keeping your code organize is important. For this session, let's create a folder under the C:\ drive.

```
c:\
└── au-2019-python
    ├── test.py
    └── <other files>
```

You can create the directory manually from using Windows Explorer or using Visual Studio Code, but you can also do this quickly from the command prompt:

```
c:\>mkdir au-2019-python
```

While you can save your files anywhere, it's important to understand the their location affects how the code is executed. The interpreter will look for the following

```
c:\>python <relative-path-to-a-python-file>
# Ie. from c:\, you need to enter the full file path
c:\>python au-2019-python/test.py
```

Alternatively, you can `cd` to where your files are and run them from there

```
c:\>cd au-2019-python
c:\au-2010-python>python test.py
```

If you try to run a file with the incorrect location, Python will tell you:

```
c:\>python test.py
python: can't open file 'test.py': [Errno 2] No such file or directory
```

Using Python Interpreter Interactively

The Python interpreter is the core of the language. It's binary application that can take Python source code and executed.

The interpreter is an invaluable tool for testing code samples and exploring the language with interactive feedback.

For this session we will run the interpreter and then execute python code one line at the time. To start a new interpreter session just type `python`

Note the `>>>` caret indicates the code is being run from an interpreter in interactive mode.In some examples, blank lines might follow to indicate the output or result of the line executed.

```
c:\au-2010-python> python
Python 3.7.3 (default, Jun  4 2019, 11:29:12)
[Clang 11.0.0 (clang-1100.0.20.17)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
```

It can be used in two ways:

- Type commands interactively using the interactive interpreter
- Execute python source code that has been saved into a file:

eg. `c:\python test.py` will execute the source code saved in `test.py` through the python interpreter

```
# c:\au-2010-python\hello.py
print("Hello!")
```

```
C:\au-2010-python\>python hello.py
Hello!
```

Language Primitives

Now that we have a Python installation setup, an editor, and a terminal, we will dive into the language by playing with it interactively.

Print

Python programs can output values using the function `print()`. The print function takes an argument (the value inside the parenthesis) and outputs it's representation below.

```
>>> print('Hello World')
Hello World
>>> print(2 + 4)
6
```

There many other built-in functions available - for a full list see the link below
https://docs.python.org/3/library/functions.html#built-in-funcs

Commenting

A hash or pound character (#) can be used to tell the interpreter to ignore all subsequent characters in the same line. This is commonly used to add comments to yourself and others.

```
>>> # This line will not be executed
```

Basic Data Types

String

Strings are used to represent text-like objects and must be enclosed by single or double quotes.

Triple quoted strings (single or double) can be used to represent multiline text.

```
>>> print("abc")
abc
>>> print('another string')
another string
>>> type("abc")
<class 'str'>
```

Integer

Positive or negative whole numbers with no decimal points.

```
>>> print(5)
5
>>> print(-10)
-10
```

Float

Positive or negative Integer values, with decimals separated by a dot.

```
>>> print(5.5)
```

Boolean

Valid boolean-type values are True and False. These types are case sensitive, so True is not the same as true. In the example below we use the operator `is` to compare boolean values/

```
>>> True
>>> False
>>> True is False
False
>>> True is True
True
```

List

Array-like container object that can hold a collection of data-types including other lists (nested lists). Unlike other languages, Python lists can hold different data-types mixed together, although that's often not a good idea. Lists are represented with square brackets. When lists are created, they are often immediately assigned to a variable, as in the examples below.

```
>>> numbers = [1,2,3]
>>> points = [[0,0,0], [2,3,4], [5,5,5]]
>>> objects = ['Desk', 'Chair', 'Lamp']
>>> print(objects)
['Desk', 'Chair', 'Lamp']
```

List items can be accessed using its index wrapped in square brackets. Indexes start at 0.

```
>>> objects[0]  # Retrieves item of index 0 (first)
'Desk'
```

Nested lists can be accessed in the same way.

```
>>> points[1][0]  # Item of index 1 (2nd), then first in nested list
2
```

Dictionary

Container-like objects that holds data using key-value pairs. Strings are often used as keys, but they can be of other data-types - including integers and floats. Dictionaries are represented using curly braces, with key-colon-value pairs, separated by commas.

```
>>> elevations= {'Level 1': 10.0,  'Level 2': 25.0}
>>> coordinates = {'absolute': [0,0,0], 'relative': [20,20,0]}
>>> person = {'Name': 'Mark', 'Age': 20, 'Address': '123 Street'}
```

None

None is a unique object type that’s used to represent an undefined value or the absence of a value. None is also case-sensitive.

```
>>> active_view = None
```

Alternative Constructors & Converting Types

In addition to using the literal representation of the data types ('abc' → string, 1 → int) you can also use built in functions provided for each type. Note howerver, these functions are usually use to create objects, instead, they can be used to convert, or cast.

Each type has specific behaviour and supported conversions.

```
>>> int(10)
10
>>> int(5.2)
5
>>> float(5)
5.0
>>> str(5)
"5"
>>> bool(0)
False
>>> bool(1)
True
```

Expressions and Statements

Expression are statements that make up the core syntactical structure of the language. You do not have to memorize these, but knowing them helps form a vocabulary that can be used to better understand and discuss a language.

Expressions and Operators

An expression is a combination of values and operators that can be evaluated down to a single value or object. The examples below use *operators* to combine data types.

Operators are symbol which can be used to perform an operation between 2 datatypes. The operation can return another symilar datatype (eg `2 + 2` returns integer `4`) or return a different type (`3 > 1` returns boolean `True`). Note while some operators can perform implicite type castings, not all combinations are possible.

```
>>> 2 * 2
4
>>> 'Hello ' + 'You' + "!"
'Hello You!'
>>> 4 > 2
True
>>> 3 == 3
True
>>> 3 == 3
True
>>> 5 * "a"
"aaaaa"
>>> 5 + "a"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Statements Overview

In contrast, statements are basically everything else, often associated with an action. Common statements are variable assignment (x = 2), and other flow-control features such as (if, else, for, etc).

```
>>> x = 3       # Assign the value 3 to the variable x
>>> if x == 3:  # Checks if the variable x is equal to the value 3
...    x = 5    # Assign the value 5 to the variable x
```

Assignment Statement

A single '=' symbol is used to assign a value to a variable name. This is different than two '==', which is a check for equality, as shown above.

```
>>> x = 1
>>> y = 3
>>> print(x+y)
4
```

"If" Statement (Conditional Flow)

If, else, and elif can be used to check for conditions, and control the flow of the program based on whether those conditions are met.

```
>>> x = 5
>>> if x > 3:
...     print('x is larger than 3')
... elif x == 3:
...     print('x is 3')
... else:
...     print('x is less than 3')
```

For Loop Statement (Iteration)

For Loops allow us to iterate over any *iterable elements*. Some of the most common iterables elements are Lists, Strings, and Dictionaries, :

```
>>> for letter in 'Hello':
...     print(letter)
'H' FIX
'e'
'l'
'l'
'o'

>>> for number in [1,2,3]:
...     print(letter)
1
2
3
```

White Space

One of the most controversial and loved features of Python is the obligatory use of white space. Unlike other languages which use brackets to group blocks of

code, Python uses spaced indentation. While "tab" characters are allowed, the official Python Style Guide advise use of 4x space characters instead.

Using the same example from above, note how indentation is used to control execution flow:

```
>>> x = 3
>>> if x == 3:
... #### note
...     # Only do this if x is equal to 3
... else:
...    # Do this if x is NOT EQUAL 3
>>> # Do something at the end, Regardless of the value of x

>>> for n in [1,2,3]:
...    # Execute this line 3x times - first as 1, then 2, then 3
... # After looop ends, continue execution here.
```

Functions and Methods

Functions

A function is a named block of code that performs an action. Functions can be defined using the keyword def followed by the function name and arguments. Functions are called by adding parentheses () after their name. Arguments are the values passed to a function (x and y in the example below). Functions can use the keyword return to return a value, although that is not a requirement.

```
>>> def print_text():     # Function Takes no arguments
...     print('My Text')  # Performs a function, but doesn't return any value
>>> print_text()
'My Text'
>>> def feet_to_inches(num_in_feet): # Function receives 1 argument
...     return num_in_feet * 12       # Returns the arg multiplied by 12
>>> feet = feet_to_inches(2)  # total variable will hold the value 7 returned
>>> print(feet)
24
```

Methods

Methods are functions defined and stored within an object. Many built-in data-types come with pre-defined *methods* built-in. That enables us to *call* or *apply* a function directly to an object. As you learn how to define your own data-types, you can create your own method, but for now we will stick with using methods defined by the built-in types.

In the examples below, we use the a list's `append` method to append a new item, and a string's `lower()` method to convert all characters to lower case.

```
>>> numbers = [1, 2, 3]
>>> numbers.append(4)   # appends an item to a list
[1, 2, 3, 4]
>>> 'LEVEL 01'.lower()
'level 01'
>>> 'i like to type'.capitalize()
'I like To type '
```

A list of all String Methods and List Methods are detailed in the official Python documentation.

Classes

Classes are *recipes* for custom data-types. For instance, one could create a class called `Door`. An instance of this class might have a *property* `door.is_open` that returns `True` or `False`, a property `door.width` that returns its width, and a method `door.open()` which sets it's `is_open` attribute to `True`

Classes are fairly advanced topic and further details are beyond the scope of this this course, but we will include an example below for your reference. You can read more about Python Classes on the official docs at docs.python.org/3/tutorial/classes.html

```
class Door():
    """ This class represents a Door """
```

```
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.is_open = True

    def open(self):
        self.is_open = True

    def close(self):
        self.is_open = False

>>> my_door = Door(30, 60)
>>> my_door.width
30
>>> my_door.is_open
True
>>> my_door.close()
>>> my_door.is_open
```

Imports

Imports can be used to load additional modules, classes, or functions into your script. These can come from another script you have written, or from the Python Standard Library. The standard library is a large collection of packages that is shipped with Python to extend functionality. For instance, the math module contains additional math-related tools; the time module gives you tools for working with time. You can read more about all the modules available in the Standard library on https://docs.python.org

Below are two simple examples that use the os and sys module to access functionality related to the operating system and the interpreter:

```
>>> import os          # imports the os module
>>> os.listdir('C:/')  # call listdir method to get files in directory
['Program Files', 'ProgramData', 'Users', 'Windows', 'Logs' ]
```

```
>>> from datetime import datetime
>>> print(datetime.now())
```

Note on * Imports

It's worth mentioning that although the template uses the syntax ***from Namespace import** **** , this is generally considered bad practice by the Python Community. The asterisk at the end tells Python to load *all* classes and methods inside the module. Depending on the namespace, you could be loading hundreds or even thousands of identifiers into the context of your script. This practice also has the following disadvantages:

1. Loading all modules can cause a namespace conflict. For example, if another module also contains a class with the same name, the first one loaded will be overwritten.
2. As objects loaded are used throughout our code, it's hard for other programmers to tell from which one of the Assemblies/Namespaces the object came from.

A better solution is to either load only the class you need, or to reference them using the parent module. For example, if you want to use the Circle and Vector classes, you could use

Unpacking

list or tuple unpacking allows us to deconstruct iterables by assigning them to corresponding variables.

```
>>> names = [“Taylor”, “Swift”]
>>> first_name, last_name = names
>>> print(first_name)
“Taylor”
>>>
>>> point = [0, 0, 10]
>>> x, y, z = point
>>> print(z)
10
```

~~Helpful~~ Essential Interactive Commands

The following commands are built-in functions that I find useful for debugging or inspecting code. Some are well known like `print()` while others are soldomly used (eg. `help` or even brand new (eg. `breakpoint`).

dir()

Prints all attributes of a given object. Not it also includes methods and "private" attributes that are not intended to be used by the user. `dir` provides a quick glimpse into the attributes a methods a data types or class has.

```
>>> dir(str)
[ ..., '__add__', '__class__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', ...]
```

help()

Shows the documentation for python objects and types. You can pass objects and modules directly. Note `help` opens an interactive viewer - use up/down arrows to scroll, and `q` to exit.

```
>>> help(str)
NameError: name 'python' is not defined
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
```

```
 |  ...
 |
 |  Methods defined here:
 |   capitalize(self, /)
 |      Return a capitalized version of the string.
 |
 |      More specifically, make the first character have upper case and the rest lower
 |      case.
 |  ...
```

type()

Prints the "type" of an object. This is particularly useful to inspect code interactively to explore a variable type during runtime.

```
>>> x = 5 + 2.0
>>> type(x)
<class 'float'>
```

print()

Print is the most well know of these. Simply put, it prints the "string value" of the object passed to the console.

```
>>> x = 2 + 7
>>> print(x)
9
```

breakpoint()

`breakpoint` is a new built-in function and was introduced in 3.7 - prior to this release, you could achieve the same by using the `pdb` module (python debugger).

Both commands tell the interpreter to stop code execution and launch an interactive debugging prompt. This can be use to inspect variables, function arguments, call functions, and even step-through your code.

You can ask pdb print a list of available commands by typing `?` or `help`. The most commonly used ones are:

- `q` or `quit` - terminate the debugger and the program execution
- `n` or `next` - go to the next line
- `c` or `continue` - close the debugger and continue program execution
- `l` or `list` - print the source code indicating where execution stopped.
- Other helpful commands I use often are `step`, `return`, `up`, and `down`,

```
>>> for x in [1,2,3,4,5]:
...     if x > 3:
...     breakpoint()

(Pdb) help

Documented commands (type help <topic>):
========================================
EOF     c           d         h          list       q         rv       undisplay
a       cl          debug     help       ll         quit      s        unt
alias   clear       disable   ignore     longlist   r         source   until
args    commands    display   interact   n          restart   step     up
b       condition   down      j          next       return    tbreak   w
break   cont        enable    jump       p          retval    u        whatis
bt      continue    exit      l          pp         run       unalias  where

Miscellaneous help topics:
==========================
exec  pdb
```

Hands on Project

For the hands on project, we will try to achieve the following:

1. Crawl files in a given sub-directory
2. Check file sizes
3. Convert file size from bytes to megabytes
4. Create list with files larger than 5mb
5. Email list of large files

Source Code

https://github.com/gtalarico/au-2019-python