Python
A Gentle Introduction

Katie Wolf → kwolf9@fordham.edu

Link to slides: http://fordh.am/iK

Link to GitHub with example code: https://github.com/katiewolf57/NYCDH-Week

How to Get Started: A Text Editor

- A text editor is an important part of writing code - it’s where you write!
- You can also use something called an IDE (Integrated Development Environment) - that’s software that has a lot of tools wrapped up into an easy to use GUI (Graphical User Interface). Typically they’ll have a place to write code, a place to implement it, and other features that make coding easier.
- It’s easy to switch between these, so don’t feel the need to commit to one just yet.
- You can start out with a very basic text editor and a command line interface and code just fine!
- Recommended: SublimeText (TE), Atom (TE) or Atom-IDE (IDE), IDLE (IDE) for Windows, Visual Studio Code (IDE), anything you’re already comfortable with!

How to Get Started: Command Line Interface

- A command line interface is where you can run your python scripts (especially if you're not using an IDE) and also run interactive sessions, which is running python code without having to write and save a .py file.

How to Get Started: Command Line Interface (Windows)

- You have Powershell and CMD readily available on Windows
  - CMD - search for “cmd” in the search bar OR press Windows+X to open the Power Users window and select “Command Prompt” or “Command Prompt (Admin)”
  - Powershell - search for “powershell” in the search bar OR press Windows+X to open the Power Users window and select “Windows PowerShell” or “Windows PowerShell (Admin)”
- I generally recommend Powershell but use whatever you find easier

How to Get Started: Command Line Interface (MacOS)

- Terminal is the best and easiest option on MacOSes.
    - Open Spotlight Search (press Command+Space or click the magnifying glass in the icon bar at the top of the screen). Search “terminal.app” and you’ll find the terminal icon.
      OR
    - Open up launchpad, search “terminal” and the terminal will icon will pop up
- I recommend pinning Terminal to your Dock for faster access!

How to Get Started

Python can be as easy to set up as you’d like!

- There are a lot of different ways to set up your programming environment, and it can be easy to get a bit overwhelmed
- Don’t worry too much about getting it exactly right the first time! If you have questions about what set up is right for you, I’ll try to steer you in the right direction during the Q&A portion.

How to Get Started: MacOS Plain & Simple

- Check to see if you already have Python installed. If your Mac is a bit older, you'll probably have a 2.x version floating around somewhere! We want to make sure we're using Python3, as Python2 isn't supported any more
  - To do this, open up terminal and type: `python –version`
  - If you see Python 2.x try this to see if you have Python 3.x: `python3 –version`
- If you don't have Python 3.x then let's install it!
- Head to the Python Download page for MacOS
- Choose the latest release of Python and follow the instructions

How to Get Started: Windows Plain & Simple

- Check to see if you already have Python installed. You'll need PowerShell for this. Right-click the Start button and select Windows PowerShell or Windows PowerShell (Admin).
  - Open up CMD or Powershell and type: python
  - If you see Python 2.x try this to see if you have Python 3.x: python3
- If you don't have Python 3.x then let's install it!
- Head to the Python Download page for Windows
- Choose the latest release of Python and follow the instructions. At the end of the installation, when given the option, it is important that you check the box that says "Add Python 3.x to PATH"

You can also set up a more tailored environment for Python!

How to Get Started: Homebrew

Homebrew installs things you need! It's an easy way to set up Python, and it's also a great tool for installing other modules or libraries. It also installs PIP which is useful for getting Python libraries.

- Easiest on MacOS - can be done on Windows but a bit more involved
- Open Source
- Easy set up

How to install:

- Open up Terminal and paste the following:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- Once homebrew is installed, type the following into Terminal

```
$ brew install python
```

How to Get Started: Anaconda

Anaconda is a package manager and Python and R distribution - so it’s a great way to get a data science oriented programming environment with little hassle. But you kind of have to commit to it! Use this is you want a pre-built set of packages and are interested in the more data-y side of Python! This also installs PIP which is great.

- Individual version is free, but other versions are not
- Works for a variety of operating systems
- A bit more involved of a set up, but you get a lot more

How to install:

- Head to their installation page and follow the instructions for you operating system

Disclaimer: These are just a few common options!

You can also install without any fuss and reevaluate your set up once you're more comfortable with Python and managing your environment!

What is Python?

- General use programming language
  - Not always the first choice but almost always the second choice!
- Open Source
- Available for all platforms/operating systems
- Easy to customize
- Home to a large community of helpful people!

More About Python

- It has a lot of documentation available
  - That includes official Beginner’s Guides
- It’s an “interpreted language”
  - This means it does what we tell it to!
- It’s “object-oriented” - mostly
  - Python has “classes” of “objects” - and almost everything is an object
- Python was made to be readable! That’s good news.

*Check the end of these slides for more resources!

What can Python do?

A lot! It can:

- Interact with the web (scrape websites, parse HTML, and a lot more!)
- Work with data in a variety of formats - it’s great for clean up, visualization, and editing/creating files
- It’s great for automating daily tasks - if you find yourself doing a task over and over, ask yourself if Python can help you out!

Some Basics: Variables

Variables store information! You can assign almost any data type to a variable. They:

- Must with a letter or an _underscore_
- CANNOT start with a number
- Must contain only letters, numbers, and underscores - no special characters or spaces!
- Are case sensitive

`A_variable = “hello”` ← All good!

`5variables = “hello”` ← Nope!

`vari&ble = “hello”` ← No!

`VARIABLE = “hello”` ← Yes!

`variable5 = “hello”` ← OK!

`My Variable = “hello”` ← Nah!

Some Basics: Data Types

Python has a lot of data types:

- Strings, integers, and floats
- Operators
- Lists and tuples
- Dictionaries
- Even more!

Some Basics: Strings, Integers, and Floats

Very basic building blocks!

Strings: “Surrounded by quotes”, loopable, countable, a sliceable array (you can pick out individual objects from a string)

Integers: Whole numbers, negative or positive

Floats: Floating Point Number, has one or more decimals, can be positive or negative

Some Basics: Operators

Used to perform operations on variables or values!

Arithmetic: `+, -, =, *, /, %, **, //`

Assignments: `=, +=, -=, *=, /=, %=, //=, **=`, etc.

Comparisons: `==, !=, >, <, >=, <=`

Logical, Identity, and Membership: `and, or, not, is, is not, in, not in`

Some Basics: Lists and Tuples

Lists store multiple items in a single variable and use [square brackets]. They are *ordered, changeable, and can have duplicates*. They are also *indexed* starting at [0].

Lists look like this: **`a_list= [“zero”, “one”, 2, True, “cat”, 2]`**

Tuples also store multiple variables but they use (round brackets). They are *ordered, unchangeable, and can have duplicates*. They are also *indexed* starting at [0].

Tuples look like this: **`a_tuple= (0, “one”, False, “one”, “dog”)`**

Some Basics: Dictionaries

Dictionaries store multiple items in key:value pairs and use {curly brackets}. They are *ordered, changeable, and don’t allow duplicates.* Note: ***dictionaries*** are only ordered in Python versions 3.7 or later!

Dictionaries look like this:

```
a_dictionary = {“name”: “Spot”,
                “species”: “Dog”,
                “age”: 7,
                “colors”: [“yellow”, “brown”, “white”]
                }
```

Now let’s actually try some stuff out!

If you have any questions or want to test something out feel free to put it in the chat!

Indexing

- Indexing allows you to access individual elements in your array items (strings, lists, tuples).
- Indexing is done with [square brackets].
- All indexes start at 0 - so [0, 1, 2, 3, ...] is how the count works
- You can use negative values to count backwards!

```
a_list = [‘cat’,  ‘dog’, ‘bird’, ‘mouse’]
a_list[0] -> ‘cat’
a_list[2] -> ‘bird’
a_list[-1] -> ‘mouse’
```
note -1 is the LAST item in the list

Slicing

- Slicing allows you to access individual elements or groups of element in your array items (strings, lists, tuples).
- Slicing is also done with [square brackets].
- The slicing format is ***[starting index : ending index : step size]***
    - `[ : ending index]` ← this will start the slice at the beginning of the list
    - `[ starting index : ]`← this will end the slice at the end of the list
    - The ending index is NOT inclusive - so putting a ‘4’ there will end the returned value at the THIRD position
- You can use negative values to count backwards!

```
a_list = [‘cat’,  ‘dog’, ‘bird’, ‘mouse’, ‘horse’]
a_list[:2] -> ‘cat’, ‘dog’,
a_list[2:] -> ‘bird’, ‘mouse’, ‘horse’
a_list[1:4] -> ‘dog’, ‘bird’, ‘mouse’
a_list[-3:-1] -> ‘bird’, ‘mouse’
a_list[0:3:2] -> ‘cat’, ‘bird’
```

Counting

You can count the length of an item (string, list, tuple, etc.) using **`len()`**! **`Len()`** stands for *length*.

This is a helpful thing to remember and often comes in handy.

Here's the formatting (also called the *syntax*):

```
a_list = [1, 2, 3, 4]
len(a_list)
>> 4
```

<- this **>>** symbol indicates what will be returned, or printed out by Python if this code is run.

Let’s try out our new indexing and slicing skills!

Fun String Things

Strings are pretty mutable and you can do a lot with them!

- Check to see if strings contain other strings using **`in`** and **`not in`**
- Use ***`upper()`*** to return the string in UPPER CASE
- Use ***`lower()`*** to return the string in lower case
- Remove whitespace with ***`strip()`***
- Replace certain letters or sub-strings with ***`replace()`***
- Split strings at certain characters using ***`split()`***
    - Ex: **`a = “hello, world!”`**
      `print(a.split(“,”))` -> [“hello”, “world”]

Note: all of the above are ***methods*** - these are built in tools that are associated with different objects. The above methods are “string methods”

Fun String Things

- Put strings together using the + character

  -
    ```
    x = “hello”
    y = “there”
    z = x + y
    print(z) -> “hello there”
    ```

  -
    ```
    a = “dogs”
    b = “cats
    c = a + “and” + b
    print(c) -> “dogs and cats”
    ```

Note! You can only combine strings with strings - no numbers allowed!

This is only the beginning of string manipulation!

Python documentation on string methods

W3 documentation on string methods

Let’s Look at Lists

Lists are malleable! You can add, remove, and change list items!

- The method ***append()*** adds things to end of lists
- The method ***insert(index, item)*** will insert an item at the specified index
- The method ***remove()*** removes things from lists
    - Use ***pop(index)*** or ***del*** to remove a specific index
- Use ***clear()*** to empty the list completely
- Use ***sort()*** to sort a list alphabetically or numerically ascending
    - Use ***sort(reverse=True)*** to sort descending
- ***Copy()*** will copy a list to a new variable
- You can join lists together in a few different ways
    - The **+** operator will add two lists together
    - The method ***extend()*** will add together ANY iterable objects

There’s a lot more to lists!

Python documentation on list methods

W3 documentation on list methods

Now What About Dictionaries?

Remember, ***dictionaries*** *ordered, changeable, and don't allow duplicates*. They're written with {curly brackets} and have *key:value* pairs.

- **`len()`** works on dictionaries - it tells you the number of key:value pairs stored in it
- You access an item by using the *key* name in [square brackets] or using the **get()** method
  - **`sample_dict[“name”] OR sample_dict.get(“name”)`**

Now What About Dictionaries?

- Use the ***keys()*** method to see all the keys -> this is returned as a dict_keys data type
- Use the ***values()*** method to see all the values -> this also also returned as a dict_values data
- Use the ***items()*** method to get all the key:value pairs -> these are returned as tuples as a dict_item data type
- You can check if a dictionary does or does not contains a *key* by using **in** or **not in**

Now What About Dictionaries?

You can change a dictionary!

- Use **update()** to change the value of a key:value pair
  - **`sample_dict.update({“name”: “Katie”})`**
  - This also works for adding items to the dictionary - just use a new key name
- You can also update or add an item by indicating a key in the brackets and assigning a value - use a new key name to create a new pair
  - **`sample_dict[“age”] = 50`**
- You can remove an item by using the **pop()** method and a key name
  - **`sample_dict.pop(“age”)`**
  - **del** will do the same thing
- You can clear a dictionary using **clear()**
  - **`sample_dict.clear()`** ← Now there’s nothing in it!

There’s a lot more to dictionaries!

Python documentation on string methods

W3 documentation on string methods

A Bit More Advanced: If … Else

If statements are a great way to actually get work done with Python. These *statements* use logical conditions (equals, not equals, less than, greater than, etc).

Syntax of an if statement:

This semicolon indicates that everything indented and under it is a part of the same block of code.

```
a = 5
b = 10
if a < b:
    print(“a is less than b”)
```

<- notice that this line is indented!

Semicolons and indentation are some of the fundamental parts of Python syntax. It’s how blocks of code inside of an ***if statement*** or ***loop*** are separated from the rest of the code.

A Bit More Advanced: If … Else

There are a few more elements to ***if statements***.

- The ***elif*** keyword means “if what I asked for in the original ***if statement*** isn’t true, then try this instead

```
if b > a:
    print(“b is greater than a”)
elif b < a:
    print(“b is less than a”)
```

- The ***else*** keyword will catch anything not considered true by any previous ***if*** or ***elif statements***. Note that you don’t follow ***else*** with any conditions or arguements

```
if b > a:
    print(“b is greater than a”)
elif b <  a:
    print(“b is less than a”)
else:
    print(“a is equal to b”)
```

A Bit More Advanced: If … Else

There are a few more important keywords: ***and***, ***or***, *and* ***pass***

- **AND** - a logical operator that means what you think it does! It allows you to test if two conditions are true at once
- **OR** - another logical operator that you can probably guess the meaning of! It allow you to test whether one of two statements is true.
- **PASS** - ***if statements*** can't be empty, but sometimes you don't want them to do anything. **Pass** fills that spot for you!

```
if a < b and b < c:
    print(“a < b and b < c”)
elif b == a or a == c:
    pass
else:
    print(“a > b”)
```

Let’s try take if…else for a test drive

A Bit More Advanced: While Loops

The ***while loop*** is one of two basic loops Python understands.
While loops perform a task WHILE a statement is true and stops once that statement becomes false (or once we tell it to stop).
Here's the syntax of a while loop:

```
x = 5
while x < 10:
    print(x)
    x = x + 1
```

notice the **:** here as well!

This is a basic counter - it is saying whatever x was, it's now equal to one more than that. This can also be written as: `x += 1`

The above loop will keep printing out x and then adding 1 to x until x reaches 10 and then it will stop!

A Bit More Advanced: While Loops

Some important keywords for ***while loops***:
break, ***continue***, ***else***

- **BREAK** - this stops the while loop even if the original condition is still true. This is usually paired with an ***if statement***.
- **CONTINUE** - causes the loop to jump back to the beginning, and skip everything following the continue. I consider this a "skip this turn" word.
- **ELSE** - much like with with the ***if statement*** this will run a block of code if the original condition is NOT true.

```
x = 1
while x < 10:
    x += 1
    if x == 3:
        continue
    if x == 6:
        break
    print(x)
else:
    print("x >= 10")
```

```
>> 2
>> 4
>> 5
```

Stay a while!

A Bit More Advanced: For Loops

Just like ***while loops***, ***for loops*** go over a set of instructions under a certain condition. However, ***for loops*** are used for *iterating* over a *sequence*. The sequence can be a list, a tuple, a dictionary, a string! It stops once it's reached the end of the sequence.
Syntax of a ***for loop***:

```
ani_list = [“cat”, “dog”, “rat”]
for x in ani_list:
    print(x)

>> cat
>> dog
>> mouse
```

```
welcome_string = “hi!”
for x in welcome_string:
    print(x)

>> h
>> i
>> !
```

A Bit More Advanced: For Loops

Some important keywords for ***for loops***: ***break***, ***continue***, ***else***

- **BREAK** - this stops the ***for loop*** even if the original condition is still true. This is usually paired with an ***if statement***.
- **CONTINUE** - causes the loop to jump back to the beginning, and skip everything following the continue. I consider this a “skip this turn” word.
- **PASS** - by Python law ***for loops*** can’t be empty, but sometimes you don’t want them to do anything. **Pass** fills that spot for you!
- **ELSE** - with the ***for loop*** it’s a bit different - ***else*** indicates a block of code to be executed when the loop is finished.

```
a_list = [1, 2, 3, 4, 5]
for x in a_list:
    if x == 2:
        continue
    if x == 4:
        break
    print(x)

>> 1
>> 3
```

Pit Stop: Continue vs. Pass

Continue and **pass** may seem similar - neither prints out anything or performs any obvious action.

- **Continue** will *go directly back to the beginning of the loop*. It will skip over everything that comes after it

- **Pass** will *not skip over anything after it*. It continues the loop and as if nothing happened. The loop will repeat if all other conditions are met

```
a_list = [1, 2, 3, 4]
for x in a_list:
    if x == 2:
        continue
    print(x)
>> 1
>> 3
>> 4
```

```
a_list = [1, 2, 3, 4]
for x in a_list:
    if x == 2:
        pass
    print(x)
>> 1
>> 2
>> 3
>> 4
```

A Bit More Advanced: For Loops

A few more useful bits!

- Use ***range()*** to loop over a range of numbers. The format is ***range(start, stop, step)***.
    - Default start is 0, you have to input a stop, and default step is 1 - ***range(0, 5, 1)*** is the same as ***range(5)***
    - Just like indexing, this is NOT inclusive

- **For loops** can be nested! This is very common. It allow you to loop over multiple things at a time. There is the ***outer loop*** (the first one, least indented) and the **inner loop(s)** (the second, more indented one).
  Each **inner loop** is executed once for each iteration of the **outer loop.**

```
for x in range(3):
     print(x)
>> 1
>>2
```

```
list_1 = [“dog”, “cat”]
list_2 = [“brown”, “soft”]

for pet in list_1:
     for adj in list_2:
          print(“The ”, pet, “ is ”, adj)
>> The dog is brown
>> The dog is soft
>> The cat is brown
>> The cat is soft
```

Going through some loops

A Bit More Advanced: Working with Files

Python is very capable of working with files!

- The key function is ***open(filename, mode)***
  - ***Filename*** is the path to your file
  - ***Mode*** is *how* you'd like to open up the file. There are four basic modes:
    - **"r"** is **read**. This is the default value. Opens the file and let's you read, but not edit/write to the file. You'll get an error if the file doesn't exist
    - **"a"** is **append**. This will open up a file for appending (adding) and will create the file if it doesn't exist. It will add new text to the end of the file.
    - **"w"** is **write**. Opens up a file for writing and will create the file if it doesn't exist. It will truncate overwrite the existing file.
    - **"x"** is **create**. This will create the file and will give an error if the file already exists.

A Bit More Advanced: Working with Files

This is the syntax:

```
f = open(“\path\my_file.txt”, “r”)
```

Notice the quotation marks. You have to put them around the filename and the mode

Now you can use the variable **f** to interact with the file.

Some good methods for working with files:

- **read()** - returns the whole text by default, you can specific the number of characters returned by placing an integer in the (brackets)
- **readline()** - returns just one line - calling it twice will read the first two lines!

```
f = open(“text.txt”, “r”)
print(f.read())
prnt(f.read(10))
>> whole document
>> first 10 characters
```

```
f = open(“text.txt”, “r”)
print(f.readline())
print(f.readline())
>> first two lines
```

A Bit More Advanced: Working with Files

You can also write to or edit files. You just have to open it up with a different mode. Use **“a”** or **“w”** to be able to write to an existing file.

```
f = open(“text.txt”, “a”)
f = open(“text.txt”, “w”)
```

Then you can use the **write()** method to add to the bottom of the file!

```
f.write(“Adding in some words!”)
f.close()
```

You can also create a new file and write directly to it. Use **“x”**, **“a”**, or **“w”** to create a new file and write to it.

`f = open(“new_file.txt”, “x”)` <- this creates a new file called new_file.txt

A Bit More Advanced: Working with Files

What if you want to read AND write?
There are a few other modes that will allow you to do multiple things:

- **“r+”** - This will allow you to read and write. You’ll get an error if the file doesn’t exist.
- **“w+”** - This will allow you to write and read. It will truncate and overwrite the existing file. It will create a file if it doesn’t already exist.
- **“a+”** - This will allow you to read and append. It will add new information to the end of the file. It will create the file if it doesn’t already exist.

A Bit More Advanced: Working with Files

Remember to close the file! Use the **close()** method. Syntax:

`f.close()` <- this one's pretty easy!

A Bit More Advanced: Working with Files

A (potentially) better method of working with a file…

You may notice that if you open up the file the way we’ve seen in previous slides, you’ll only be able to read each line one and then it’s inaccessible by future code.

Using the ***with statement*** avoids that! Syntax:

```
with open(“my_file.txt”, “w”) as f:
    f.write(“hello world!”)
    f.read()
    f.readlines()
```

A Bit More Advanced: Working with Files

One more useful method:

```
with open(“my_file.txt”, “a+”) as f
    f.write(“new line at the end of the file”)
    f.seek(0)
    print(f.read())
```

This will send the reader back to the top of the file

This sends to the reader to the very end of the file. If you tried to read the file out now, you’d get only a blank, because the reader has nothing left to read.

Some Other Good Things to Know: Modules/Libraries/Packages

Modules, libraries, and packages are ready to go, pre-built libraries of code.
If you're struggling to figure out how to code something from scratch, there's probably a module/library/package for it. They're all slightly different, but work similarly.
You can also create your own libraries, but that's not what we're covering here!

To use a one of these, you *import* it - this is usually the first bit of code you see.

`import random` <- random is a module used to generate random numbers

You can also import them under an easier name to use

`import pandas as pd` <- Pandas is a great library for working with data

Note: you often have to install these before using! Use PIP for this! Python also comes with some default ones!

Some Other Good Things to Know: Modules/Libraries/Packages

Once you've imported your module/library/package, you call it using the name you imported it under!

Syntax:

```
import random
print(random.randint(0,6)) <- here I'm calling a module and using one of
                                the methods that comes with it

import pandas as pd
my_dataframe = pd.DataFrame(my_dataset)
```

Modules/Libraries/Packages to know about!

- **Pandas Library**- great for working with databases/dataframes, CSVs, and JSON files
  - Great for cleaning up, sorting, and plotting data
  - Tutorials: W3 Schools, List of tutorials recommended by Panda devs
- **CSV Module** - a default module from Python, great for CSV work if you don't want to get into Pandas just yet
  - Allows you to read, edit, and create
  - Tutorials: W3 Schools, Geeks for Geeks
- **BeautifulSoup** - a Python package for parsing HTML and XML.
  - If you're interested in web-scraping, know this one!
  - Created by someone currently at NYPL, so you know it's cool!
  - Tutorials: Real Python, Geeks for Geeks, ACRL Tutorial
- **Matplotlib** - If you want to visualize data this is a great place to start
  - Works with a variety of other libraries, including Pandas
  - Tutorials: Matplotlib Documentation, W3 Schools, Geeks for Geeks

Some Other Good Things to Know: Functions

A function is a "block" of code that will run when you call it. Basically functions are reusable sets of instructions that return some type of result. You *define* them using the `def` *keyword*.

Functions look like this:

```
def a_print_fucntion(x):
    print(x)

a_print_function("hello") -> hello
```

These will come in handy once you start creating longer programs. If you notice you're implementing the same instructions over and over, consider turning it into a function.

Explore Further

Free Resources

- W3 Schools
- Google Schools
- Library Carpentry
- Python Beginners Documentation
- Automate the Boring Stuff

Not free but still good

- LinkedIn Learning/Lynda (available via NYPL)
- Learn Python the Hard Way
- O'Reilly Books