Introduction to Python

A HMS Research Computing and DevOps Production

rchelp@hms.harvard.edu

Introduction

Why python?

Python Is...

simple - resembles plain English

easy - no need to declare (in most cases), memory management

clean - whitespace-formatted for visibility

interpreted - good for developing, bad for performance (more on this later)

Interpreted	Compiled
Rapid prototyping	Faster Performance
Requires interpreter	Requires compiler (GCC, etc.)
Dynamic typing	Static typing
Code-level optimization	Code- and Compiler-level optimization

Data and Script Management

Data Management

As you run more jobs, you'll probably end up creating a whole bunch of files. In the same way it's important to plan bench projects beforehand, it's good to think early about how you should manage and organize all those files you'll be making. *Note: be sure to ask your PI and your department about standard practices in your field!*

Harvard Biomedical Data Management Website: https://datamanagement.hms.harvard.edu/
Resources & Information: https://datamanagement.hms.harvard.edu/overview

Top Data Management Best Practices

- **Planning**: Document the activities for the entire lifecycle. Create a Data Management Plan including sponsorship requirements, realistic budget, assigned responsibilities, all the data to be collected, *and each of these topics*! https://datamanagement.hms.harvard.edu/planning-overview

- **Organization**: Define how the data will be organized, including what is your folder hierarchy and how did you get from raw data to the final product? Consider versioning control for changes for both software and data products. https://datamanagement.hms.harvard.edu/versioning-1

- **Documentation**: Explain how the data will be documented such as naming conventions, acronyms, data fields and units. Determine whether there is a community-based metadata standard that can be adopted. Create a README file to record the metadata that will be associated with data. https://datamanagement.hms.harvard.edu/readme-files

Top Data Management Best Practices

- **Storage**: Your storage plan is integral to data management. Consider how the data will be stored and protected over the duration of the project. Identify short-term and long-term storage options. Remember to link accompanying metadata and related code and algorithms. https://datamanagement.hms.harvard.edu/storage-overview

- **Sharing**: Describe what data will be disseminated, to who, when, and where. Identify sharing tools to work with collaborators during the project and publish data in an open repository. Be sure to use standard, nonproprietary approaches and provide accompanying metadata & associated code. https://datamanagement.hms.harvard.edu/data-sharing

- **Retention**: Think about your preservation strategy from the start & adhere to your lab's standard practices. Research records should generally be retained no fewer than seven (7) years after the end of a research project or activity. https://datamanagement.hms.harvard.edu/data-retention

Accessing Python On O2

Accessing Python on O2 – Cluster Access

- Connect using terminal applications
  - Linux or Mac – Native Terminal, Putty
  - Windows – MobaXterm (Recommended), Putty
- For detailed instructions on how to login to O2, visit our wiki.
- Please Note: **All** O2 cluster logins originating from outside the HMS network will require two-factor authentication

Accessing Python on O2 – Logging In

Open a terminal and ssh into o2.hms.harvard.edu:

```
$ ssh rc_training01@o2.hms.harvard.edu
rc_training01@o2.hms.harvard.edu's password:

rc_training01@login01:~$
```

Accessing Python on O2 –Interactive job

Once on O2, start an interactive job:

```
rc_training01@login01:~$ srun --pty -p interactive -t 0-2 bash

# will get a message about your job waiting for resources
# once resources are allocated, you will be placed on a compute node and your
prompt will change:

rc_training01@compute-a:~$

# now can enter any commands you want
```

Accessing Python on O2 – Finding Python

```
$ module avail python
    No modules found!
    Use "module spider" to find all possible modules.
    Use "module keyword key1 key2 ..." to search for all possible modules matching
    any of the "keys".

$ module spider python
       Versions:
          python/2.7.12
          python/3.6.0
          python/3.7.4
```

Accessing Python on O2 – Finding Python, cont'd.

```
$ module spider python/3.6.0
    You will need to load all module(s) on any one of the lines below before the
    "python/3.6.0" module is available to load.

          gcc/6.2.0
    (snip)

$ module load gcc/6.2.0 python/3.6.0
$ which python3
/n/app/python/3.6.0/bin/python3
$ python3
Python 3.6.0 (default, May  2 2018, 17:23:32)
[GCC 6.2.0] on linux
 Type "help", "copyright", "credits" or "license" for more information.
 >>>
```

Today's course will use 3.6.0. (To exit the interpreter, type ctrl+D, exit(), or quit().)

Alternatively, virtual environments

```
$ module load gcc/6.2.0 python/3.6.0
 $ which virtualenv
 /n/app/python/3.6.0/bin/virtualenv
 $ virtualenv nameofenv --system-site-packages
New python executable in nameofenv/bin/python
 Installing setuptools, pip, wheel...done.
 $ source nameofenv/bin/activate
 (nameofenv)$ which python3
 ~/nameofenv/bin/python
```

To deactivate:

```
(nameofenv)$ deactivate
$
```

Why virtual environments?

- Python has modules. if you want to install your own, you need a virtual environment.
- The version on O2 has a certain number of built-in modules; the `--system-site-packages` flag allows your virtual environment to inherit those packages so you don't have to reinstall them yourself, if you're using python modules that are older than 3.7.4.
- For more information, you can visit our Personal Python Packages wiki page.

Installing Python Modules

- To install modules, generally use `pip3` or `easy-install` (we recommend `pip3`):

```
(nameofenv)$ pip3 install packagename
```

- If that doesn't work (e.g. you've downloaded the archive manually), follow the instructions in the provided README file, but it'll go something like

```
(nameofenv)$ python3 setup.py install
```

- (some will have you use `build` before `install`). Make sure you read the (hopefully provided) instructions when installing modules manually.

Viewing Modules

- **pip list** shows ALL packages. **pip freeze** shows packages YOU installed via **pip** command in a requirements format.

```
$ python3 -m pip freeze
aiohttp==2.3.10
alembic==1.4.2
appdirs==1.4.4
argh==0.26.2

$ python3 -m pip list
Package                 Version
----------------------- ----------
aiohttp                 2.3.10
alembic                 1.4.2
appdirs                 1.4.4
argh                    0.26.2
```

Viewing modules cont.

- To see ALL modules available on your Python build, type either of:

```
$ python3 -c "help('modules')"
```

and

```
$ python3
Python 3.6.0 (default, May  2 2018, 17:23:32)
[GCC 6.2.0] on linux2
 Type "help", "copyright", "credits" or "license" for more
information.
>>> help('modules')
```

Viewing modules cont.

- And it will output something like:

```
Please wait a moment while I gather a list of all available modules...

(there might be some warnings here...)

__future__          aifc                imaplib             scipy
_ast                antigravity         imghdr              secrets
_asyncio            argparse            imp                 select
_bisect             array               importlib           selectors
.
.
.
```

Let's Learn Python!

Basic Syntax

- Statements are terminated by newlines (e.g. enter key)
- Examples:

```
>>> a = 1
>>> print("hello world")
 hello world
```

Nuances: Quotations

- In general, double and single quotes are interchangeable, both for printing and argument passing:

```
>>> print("hello world")
hello world
 >>> print('hello world')
 hello world

 >>> str = 'abcdefg'
>>> str1 = "abcdefg"
>>> str == str1
 True
```

Nuances: Escaped Characters

- Sometimes, you'll need to escape (backslash) certain special characters to get them to display correctly when printing. For example, if you want to preserve double quotations in your string:

```
>>> print(""hello world"")
 File "", line 1
    print(""hello world"")
                      ^
 SyntaxError: invalid syntax
>>> print("\"hello world\"")
 "hello world"
```

- A list of escape characters can be found at the Python documentation and elsewhere on the internet.

Comments

- Comments are used to make code more legible. In python, they are denoted with the octothorpe:

```
#!/usr/bin/env python3
 ...
 # do stuff here
 ...
```

- Multi-line comments can either be manually be broken down, or held in a docstring with triple quotes:

```
"""
 this is a
multi-line comment
 """
```

- We'll revisit the `#!` line in a little bit.

Basic Features

DATA STRUCTURES, LOOPING, ETC.

Data Types

- Python is dynamically typed; there is no need to declare (e.g. `int n = num`). You can also reassign:

```
>>> number = 1
>>> number
 1
>>> number = 2
>>> number
2
>>> str = 'abc'
>>> str
'abc'
 >>> str = 'def'
>>> str
'def'
 >>> str = 1
 >>> str
 1
```

Data Types, an aside

- However, strings (and some other stuff like tuples) are not *mutable*; you cannot modify its content. When you reassign the variable as seen previously, you're actually generating a new object. You can't modify the original object:

```
>>> s = "abc"
 >>> s[0]
 'a'
 >>> s[0] = "o"
 Traceback (most recent call last):
    File "", line 1, in
 TypeError: 'str' object does not support item assignment
```

- The above example was shamelessly lifted off Stack Overflow.

Back to data types

- Like most languages, Python has higher level data structures. Of note are lists (arrays) and dictionaries. Relatedly, there are also tuples and sets. Each structure fills different niches.

Lists

- Lists are the most versatile; they are the effective equivalent of arrays in other languages. They are ordered, and you can fetch specific indices. You can mix and match types, have duplicates, nest lists, etc. To generate a list:

```
>>> lst = [] #initialize a list of indeterminate size
 >>> lst.append('a')
 >>> lst
 ['a']
 >>>lst.extend(['b', 'c'])
 >>>lst
 ['a', 'b', 'c']

 >>> lst2 = [None]*5 #initialize a list of predetermined size (an array)
 >>> lst2
 [None, None, None, None, None]
 >>> lst2[1] = 'foo'
 >>>lst2[1]
 'foo'
 >>> lst2
 [None, 'foo', None, None, None]
```

Lists, cont.

```
>>> lst.insert(2, 'new_element')
>>> lst
['a', 'b', 'new_element', 'c']
>>> lst.pop(2)
'new_element'
>>> lst
['a', 'b', 'c']
>>> lst.pop()   # if no argument provided, pops last element
'c'
>>> lst
['a', 'b']
```

Basic list manipulations

- lists are zero-indexed (count from 0), and you can reference positions like so:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1]
 2
 >>> lst[1:]     # sublist from second entry to end
 [2, 3, 4, 5]
 >>> lst[1:3]    # note that the fourth element is omitted; when end indices are specified, Python
stops BEFORE they are reached
 [2, 3]
 >>> lst[:3]     # sublist from beginning to fourth entry
 [1, 2, 3]
 >>> lst[1::2]  # sublist from second to end, every other element
 [2, 4]
 >>> lst[-1]     # last entry (reverse indexing)
 5
 >>> lst2 = [1, 2, 3, [4, 5]]    # nested list
 >>> lst2[3][1]  # specify all relevant indices from outside in
 5
```

Sets

- Sets are lists that do not allow duplicates. Sets are useful if you need to take unions, intersections, etc. To generate sets:

```
>>> st = set(lst)
 >>> st
 {'a', 'c', 'b'}

 >>> lst3 = [1, 1, 2, 3, 4, 4]
 >>> st2 = set(lst3)
 >>> st2
 {1, 2, 3, 4}

 >>> st3 = {1, 2, 3}
 >>> st3
 {1, 2, 3}
```

Sample set operations

- Here are some elementary operations you can perform with sets:

```
>>> set1 = {1, 2, 3}
 >>> set2 = {3, 4, 5}
 >>> set1 | set2          # union
 {1, 2, 3, 4, 5}
 >>> set1 & set2          # intersection
 {3}
 >>> set1 - set2          # set difference
 {1, 2}
 >>> set2 - set1
 {4, 5}
 >>> set1 ^ set2          # symmetric difference
 {1, 2, 4, 5}
 >>> set2 ^ set1
 {1, 2, 4, 5}
```

- More information can be found in the documentation.

Tuples

- Tuples are lists, but immutable. Once created, they cannot be modified. If you know the size of your data structure, tuples are preferred over lists because Python will know exactly how much memory to allocate. To create tuples:

```
>>> empty_tuple = ()
>>> empty_tuple
()
>>> one_element = 1,
>>> one_element
(1,)
>>> mixed_tuple = (1, 'a', [1, 'a'])
>>> mixed_tuple
(1, 'a', [1, 'a'])
>>> lst = [1, 2, 3]
>>> tuple_from_list = tuple(lst)
>>> tuple_from_list
(1, 2, 3)
```

Dictionaries

- Dictionaries are associative collections (maps). They are composed of key:value pairs (most of the time). To create a dictionary:

```
>>> empty_dict = {}    # careful not to confuse this with empty sets; to initialize an empty
set, use set())
>>> empty_dict
 {}
 >>> dict1 = {1: 'a', 2: 'b', 3: 'c'}
 >>> dict1
{1: 'a', 2: 'b', 3: 'c'}
 >>> dict2 = dict([(1, 'a'), (2, 'b'), (3, 'c')])
 >>> dict2
 {1: 'a', 2: 'b', 3: 'c'}
 >>> dict3 = dict(a=1, b=2, c=3)     # only works if keys are strings
 >>> dict3
 {'a': 1, 'c': 3, 'b': 2}
 >>> dict4 = dict(1=a, 2=b, 3=c)
File "", line 1
   SyntaxError: keyword can't be an expression
```

Dictionaries, cont.

```
>>> dict1
{1: 'a', 2: 'b', 3: 'c'}

>>> dict1[4] ='d' # add to dictionary
>>> dict1
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> dict1[4] = 'e' # add to dictionary, but overwrite as key exists
>>> dict1
{1: 'a', 2: 'b', 3: 'c', 4: 'e'}
>>> dict1.update({5:'f', 6:'g'}) # add multiple key value pairs to dictionary
>>> dict1
{1: 'a', 2: 'b', 3: 'c', 4: 'e', 5: 'f', 6: 'g'}

>>> dict1.pop(1) # remove a key value pair from dictionary
'a'
>> del dict1[2] # can also use del to remove key value pair
>>> dict1
{3: 'c', 4: 'e', 5: 'f', 6: 'g'}
```

Basic dictionary manipulations

- Dictionaries don't have indices to reference, but they do have keys and values. To interact with dictionaries:

```
>>> dict1 = {1: 'a', 2: 'b', 3: 'c'}
 >>> dict1[1]
 'a'
 >>> dict1.keys()
 dict_keys([1, 2, 3])
 >>> dict1.values()
 dict_values(['a', 'b', 'c'])
>>> dict1.get(4, "I am a default value")
 "I am a default value"
```

- There's a lot more you can do with these data structures. If you find yourself wondering if *x* data structure can do *y* thing, feel free to search for an answer. Often, there will be a solution!

Flow Control

FOR, WHILE, IF/ELSE

For Loops

- For loops are very straightforward to implement, if a little obfuscated. To iterate over a list:

```
>>> things = ['apple', 'banana', 'cherry']
>>> for thing in things:
...     print(thing)
...
apple
banana
cherry
>>>
```

An aside: scoping

- Code within indented blocks is known to be restricted in *scope* to that block. That is, anything that happens within that block of code does not necessarily persist outside that block of code. We'll revisit this momentarily...

For Loops cont.

- This will go through every item in your list (or tuple or whatever) in order. If you also wanted to fetch indices, you'd use:

```
>>> for idx, thing in enumerate(things):
 ...     print(idx, thing)
 ...
 0 apple
 1 banana
 2 cherry
 >>>
```

For Loops cont.

- The `enumerate()` function is actually an *iterator* that spits out a tuple consisting of (index, value) and assigns each to idx, name. This is called a **named tuple**.
  The `for a in b` structure can be replaced with any generic construct (within reason). For a generic loop, you can do something like:

```
>>>for i in range(5):
    ...     print(i)
    ...
    0
    1
    2
    3
    4
```

An application of for loops: File I/O

- File input/output (I/O) in python is conventionally done via the creation of a *context manager*. You can think of this (i.e. the file handle and its associated operations) as a block of code that executes for as long as you need that file to stay open.

```
with open('file.txt', 'r') as f:
    for line in f:
        print(line)
```

- The code runs in the context of the open file, and automatically closes the file when done.
- This is an example of scoping. We'll have another example when scripting.

File I/O cont. (an aside)

- The first argument of open() is the filename. The second one (the mode) determines how the file gets opened.
  - Read the file, pass in "r". Read and write the file, pass in "r+"
  - Overwrite the file, pass in "w". Writing and reading the file, pass "w+" Overwrites the existing file if it already exists.
  - If you want to append to the file, pass in "a"

```
with open('file.txt', 'r', encoding="utf-8"): as f:
    f.readlines(2) # Read in N # lines
    for line in f: # Do something with each line
        x = f.readline()
```

```
import json, os
with open('file.txt', 'w', encoding="utf-8"): as f:
    for x in range(100):
        f.writeline(x)
    # write all env vars to file
    f.write(json.dumps(os.environ, indent=2))
```

If the you are getting unexpected input when performing file IO, try adding the following:

```
with open('file.txt', 'r', encoding="utf-8"): as f:
```

While Loops

- Similar in function to `for` loops, while loops are used to iterate, and are useful if you don't know how long to iterate for (e.g. searching for convergence, etc.). A generic while loop looks like this:

```
>>>toggle = True
 >>> while toggle == True:
...     toggle = False
 ...     print(toggle)
 ...
 False
```

- A very simplistic example, but the above while loop runs one iteration, then exits because `toggle` is no longer `True`.

If/elif/else

- `if/else` statements are useful when you need to handle different cases in your workflow. A generic if/elif/else statement structure:

```
# take some input (let's say, a number)
if input == 0:       #if input is zero
    print('zero')
elif input % 2 == 0: #if input is otherwise even
    print('even')
else:                #if input is odd
    print ('odd')
```

- You may use as many `elifs` as you require to solve your problem.

An aside: try/except

- Python has the interesting distinction of relatively lightweight exception handling. Exceptions are only computationally expensive if they trigger. For more information, this page has a good analysis of if/else versus try/except.

A few pertinent modules

NumPy, SciPy, Matplotlib (in brief)

- NumPy is the go-to module if you need to perform complex arithmetic or do matrix operations (manipulate data frames). It is much more efficient than using system Python utilities.
- SciPy has a bunch of interesting functions that automate various aspects of scientific computing, like statistics and higher-level mathematics. Most of these are callable in one line. (documentation, includes NumPy)
- Matplotlib is a basic plotting module. You can generate plots in real time (X11) or create them and write to files to view later. There is also a degree of customization afforded in plots. (documentation)

SciPy

```
>>> from scipy import constants          # `from scipy import *` will not
behave as expected
 >>> constants.c                          # speed of light
299792458.0
 >>> from scipy.stats import norm
 >>> norm.cdf(5, 0, 3)                    # P(x<5) if X~N(0,3)
 0.9522096477271853
```

NumPy

```
>>> import numpy as np       # the general way; using `as` allows you to set an alias (if the name is too
long to type)
>>> cvalues = [25.3, 24.8, 26.9, 23.9]    # a typical python list of Celcius values
>>> C = np.array(cvalues)       # create a numpy array
>>> print(C)
[ 25.3  24.8  26.9  23.9]       # note that they look identical
>>> print(C * 9 / 5 + 32)       # convert to Fahrenheit using scalar multiplication
[ 77.54  76.64  80.42  75.02]
 >>> fvalues = [x*9/5 + 32 for x in cvalues]    # with a typical list, you need to loop over it and compute
element-wise instead
 >>> print(fvalues)
[77.54, 76.64, 80.42, 75.02]       # same result, less efficient/readable
```

More NumPy

- A brief look at arrays: NumPy data structures

```
>>> nested_list = [[3.4, 8.7, 9.9], [1.1, -7.8, -0.7], [4.1, 12.3, 4.8]]     # a python nested list
>>> nested_list
[[3.4, 8.7, 9.9], [1.1, -7.8, -0.7], [4.1, 12.3, 4.8]]
>>> A = np.array ([ [3.4, 8.7, 9.9], [1.1, -7.8, -0.7], [4.1, 12.3, 4.8]])   # a numpy array (matrix)
>>> A
array([[  3.4   8.7   9.9]
  [  1.1  -7.8  -0.7]
  [  4.1  12.3   4.8]])
```

- Note the formatting of the print. This is indicative of the logic and illustrates why numpy is the preferred implementation of Python matrix operations. The SciPy documentation linked previously also includes NumPy documentation.

Matplotlib

```
>>> import matplotlib.pyplot as plt
 >>> plt.plot([1,2,3,4])
 [(some memory address)]
 >>> plt.ylabel('some numbers')
(another memory address)
 >>> plt.show()
```

- You can save the image using something like `plt.savefig('image.png')`.

Object Oriented Programming

- Python can also implement classes.
- Using classes over functions in Python is generally to the user's discretion, but standard common sense and logic applies as usual. If your program expands to the point where you think an object is more effective than functions, then feel free to implement it.
- No instruction on classes will be given here, as it is beyond the scope of the course. For more information, you can consult an in-depth article like this or look straight at the Python documentation.

Before we leave the interpreter:

- When you exit the interpreter, you will lose your Python command history.
- If you want save a list of your previous commands, you can use the readline Python interface. For example:

```
>>> import readline
>>> readline.write_history_file('/home/mfk8/python_history.txt')
```

- Once you open a new session, you can import your history:

```
>>> import readline
>>> readline.read_history_file('/home/mfk8/python_history.txt')
```

- For more information, check out the Python docs.

A Brief Introduction to Scripting

Up until now, we've mostly been playing inside the interpreter. Here, we'll briefly go over what is required to write a proper Python program.

To start:

- Strictly speaking, all you need for a Python program is a text file with the shebang line on top. Recall:

```
#!/usr/bin/env python3
```

- This line indicates to the computer that this is a python program, and it should look in this location to execute. Similar shebangs may look like:

```
#!/usr/bin/python
#!/bin/bash
 #!/usr/bin/perl
 etc.
```

- The shebang is telling the computer to look in the specified directory for the proper method of execution.

Why use env?

- `env` is used for portability. On *nix machines, there is a path `/usr/bin` where most system programs/binaries are installed. If you need to install multiple versions, this can be an issue. Which version of Python do you want to use?
- This is what `env` is for. It tells the computer to look at the current environment, and choose the Python that is currently in use. This is especially helpful on O2, where we have multiple versions installed at the same time.

A basic program:

- Once you've included the shebang, you can get right to it. Start typing lines just as you would in the interpreter, and once you execute your program, each line will resolve itself in order.

```
#!/usr/bin/env python3

print("hello world!")
```

- Type this into a text file, and save it as whatever, and include `.py` at the end for your own convenience.
- To execute this program, just type at the terminal:

```
$ python3 file.py
```

- You've just written your first python program!

Basic structure:

- The previous program is not likely going to be your typical use case. More complex programs may use modules, functions, classes, etc.
- A typical program will have the following structure:

```
#!/usr/bin/env python3

# IMPORT EVERYTHING HERE

# CREATE FUNCTIONS AND/OR CLASSES HERE

 def main():
    # CODE THAT DOES NOT BELONG IN FUNCTIONS GOES HERE

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

Extending the previous example:

```
#!/usr/bin/env python3

# no modules were required, so none were imported

# no functions were needed, so none were created

def main():
    print("hello world")

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

What are functions?

- Functions are typically used when you have repetitive code. Instead of pasting the code over and over, just put it in a function, and use ("call") it when needed. For example:

```
def do_thing():
    print("hello world")
    return
```

- Then to reference it, just type elsewhere in your program:

```
...
do_thing()
...
```

- and the program will execute the code within the `do_thing` function.

Functions, cont'd:

- You can also make functions take arguments:

```
def do_thing(phrase):
    print(phrase)
    return

phrase = "hello world"
do_thing(phrase)
```

- You can also assign values to variables with functions. To do this, make use of the `return` keyword. Note that previously, it has been naked, which means nothing is returned. Python is smart enough to know what you mean if you omit `return`, but it's always nice to include it for consistency.

Extending the previous example again:

```
def do_thing(phrase):
    print(phrase)

    new_phrase = "goodbye world"

    return new_phrase

 ...

phrase = "hello world"

phrase2 = do_thing(phrase)
print(phrase2)
```

Putting it all together:

```
#!/usr/bin/env python3

 def do_thing(phrase):
    """Transform the input phrase into a new phrase."""
    print(phrase)
    new_phrase = "goodbye world"

    return new_phrase

 def main():
    phrase = "hello world"
    phrase2 = do_thing(phrase)

    print(phrase2)

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

Compatibility

- Today's course was taught on 3.6.0.
- There are many versions of Python available, and not all are created equal. Most distinct is the difference between 2.x and 3.x; there are several syntax changes that will be required to use version 3.x.
- 2.x has reached the end of its support life, it is recommended that 3.x is used. More information on 2.x Sunset here.
- For more information, look at this trusty 2.x to 3.x compatibility cheat sheet.

Thanks for coming!

If you have any questions, feel free to email us at rchelp@hms.harvard.edu or visit our website to submit a ticket. We also do consulting!

Please take the course survey!

- Accessible through the Harvard Training Portal
  - https://trainingportal.harvard.edu/

- Click on “Me” then “Intro to Python”
- Scroll to “Evaluations” and click on the survey
- We appreciate any feedback or comments!