Python for Scientific Data Analysis

Basic Python

Section 1: Printing and Variables

Hello World

The first step in learning Python or any other language is to ask a piece of code to print a statement.

To do this from command line, first you type `python` in your terminal window, which will bring up the Python prompt.

Printing with statements with Python 3 are enclose by parentheses.

E.g.

`print(hello world)`

will print ...

```
print("hello world")
```

```
hello world
```

Note: if you do not enclose this string with ", then it will trigger a traceback error because *hello world* is not a named variable. If you try to treat *hello world* as a variable you will get a traceback error: e.g. `>>> hello world = 5` triggers an error. However, the following does not trigger an error:

```
hello_world = 7
hello_world='yo'
hello,world = 7,11
hello_world = 'yo','checkit'

print(hello_world)
print(hello,world)
```

```
('yo', 'checkit')
7 11
```

where then you type `print(hello_world)` or `print(hello,world)` at the prompt.

What happened in these four examples? In the first two, we declared a single variable, `hello_world`, as an integer 7 or a string `yo`. But Python allows you to declare multiple variables in a single line. This is shown in third and fourth example: here, `hello` and `world` are two separate variables simultaneously set to two different strings. Cool, right?

The example code ex1.py shows demonstrates how this looks within a piece of code.

To execute, type `python ex1.py` in your Terminal window from within that subdirectory. Or, from the main directory, you can import it, and it will run automatically

```
import code_ipynb.ex1
#note: this will only print out the first time!
```

```
hello world
yo
5
yo checkit
7 11
```

You should see the following:

```
hello world
yo
5
yo checkit
7 11
```

Variables and Printing

Okay, now we are going to make things slightly more complicated. When you print something with Python (or any other language), you want to control the format of what you write: do you want to write a string? An integer? decimal? floating point? That's where *formatted* print comes into play.

Formatted Printing: The standard way

Python's *formatted* print statements come after a `%` for a print statement: e.g. `print("my name is %[formatting statement]" % [variable]`).

Fortunately, these are pretty easy to figure out: `%s` (e.g. "yo") for string, `%d` or `%i` for decimal integers (e.g. 5), `%f` for floating point (e.g. 3.141569), `%e` for lower case exponential notation (e.g. 1.41569e+03), etc. There are a few others but these are the most important ones for now.

Now, look at ex5a.py. It should read as follows

```
my_name = 'The Great Cornholio'
my_age = 21 # ya right
my_height = 72 # with stilts
my_weight = 9 # lbs
my_jump = 15 #feet
my_eyes = 'White'

print("My name is %s." % my_name)
print("I am %d inches tall and I weigh %d pounds." % (my_height,my_weight
))
print("Even though my eyes are %s, I can jump %d feet in the air" % (my_e
yes,my_jump))
print("If you add %d, %d, and %d together you get %d.  Scared now of the
%s?" % (my_age,my_height,my_jump,my_age+my_height+my_jump,my_name))
```

If you execute this code, you should see:

```
My name is The Great Cornholio.
I am 72 inches tall and I weigh 9 pounds.
Even though my eyes are White, I can jump 15 feet in the air
If you add 21, 72, and 15 together you get 108.  Scared now of the The Gr
eat Cornholio?
```

Notice another thing. In the soure code we put #s all over the place. These look like comments, right? That they are! These do not print out when you execute the code.

Formatted Print Statements: The second, more 'modern' way

Another way to do formatted print statements are with square brackets `{ }` and a format statement. Here's an example

```
statement='far out'
print("this statement is {}, dude".format(statement))
```

```
this statement is far out, dude
```

You should see `this statement is far out, dude`.

Now, slightly more complicated

```
print("this statement is {0} {1}, dude".format(statement,'seriously'))
```

```
this statement is far out seriously, dude
```

See what we've done here? we now have two variables with formatted print.

More control over formatting

Now we will take the second formatting approach and make this more complicated ...

```
print("this statement is {0:.1s} {1:s}, dude".format(statement,'seriously
'))
```

```
this statement is f seriously, dude
```

which prints out `this statement is f seriously, dude` . See what happened? if not, look to see what happens here:

```
import numpy as np #don't worry about what this line means right now. jus
t think "command allowing us to do stuff"
eval=np.e
print("this stupid number is {0:f}, not {1:.3f} or {2:.5f}, \
but if we want more than the \
 default precision of {3:d} spaces after the decimal, \
 we can write it as something different like {4:.10f}".format(eval,eval,e
val,6,eval))
```

```
this stupid number is 2.718282, not 2.718 or 2.71828, but if we want more
 than the  default precision of 6 spaces after the decimal,  we can write
 it as something different like 2.7182818285
```

which prints out as `this stupid number is 2.718282, not 2.718 or 2.71828, but if we want more than the default precision of 6 spaces after the decimal, we can write it as something different like 2.7182818285`

Note: all of this is on one line: the `\` signifies "continue the line".

So what happened there? we can control the output. For example, in the first case, we said "format this as a floating point". In the second "format this as a floating point but only leave 3 spaces after the decimal". And so on. By default, Python 3.x prints out 6 decimal spaces for a floating point. If you want more precision, then you can just add more space, as we do in the last formatted print for the variable `eval`.

Now what if we wanted to use `%` for formatted print with all that control?. Here you go:

```
print("this stupid number is %f, not %.3f or %.5f, \
but if we want more than the \
 default precision of %d spaces after the decimal, \
 we can write it as something different like %.10f" %(eval,eval,eval,6,ev
al))
```

```
this stupid number is 2.718282, not 2.718 or 2.71828, but if we want more
 than the  default precision of 6 spaces after the decimal,  we can write
 it as something different like 2.7182818285
```

A Note on how to `run' Python code

In general, to execute Python code, there are a couple of ways:

1) type `python [name of python code]` in your Terminal if the code is also in your local working directory (or is a script callable from anywhere on your machine ... don't worry about that for now)

2) in interactive mode where you have to import a module from a piece of Python code and then do something: this works so long as the piece of code is in your Python path (either local or within your path variable). e.g.

from Terminal, you first type `python` (or, in my case, python3). "Running" python then has the following structure (note, each line is a separate line in the interactive mode ... i.e. a separate ">>>")

```
>>> import [name of function]

>>> [variable] = [function].[property]
```

An example ...

```
>>> import numpy as np #we will encounter this "numpy" library a lot!
>>> d_tel = 8.2 #telescope diameter (meters)
>>> lambda= 1.6 #wavelength (microns)
>>> fwhm = 1.028*206265*lambda*1e-6/d_tel
```

3) A 3rd way is accessing Python code through Jupyter notebooks. You first need to split off a "cell" in the notebook specifically for this calculation and then execute. Here's how to execute the above code through notebooks.

```
import numpy as np
d_tel = 8.2 #telescope diameter (meters)
lam_tel= np.array([1.25,1.65,2.16]) #wavelength (microns) note: we don't
call it lambda because that is a separate, canned Python thing
fwhm = 1.028*206265*lam_tel*1e-6/d_tel

for i in range(0,len(lam_tel)):
 print("The telescope (diffraction-limited) FWHM in arcseconds given a di
ameter of {0:.1f} meters \
 and a wavelength of {1:.2f} microns is {2:.4f}".format(d_tel,lam_tel[i],
fwhm[i]))
```

```
The telescope (diffraction-limited) FWHM in arcseconds given a diameter o
f 8.2 meters and a wavelength of 1.6 microns is 0.0414
```

Option #1 is the simplest but gets a bit unwieldy for anything you will want to use for research. Option #2 is often the way through professional code but can be hard to visualize. Option #3 is great for visualization and can handle complex code that you write elsewhere and then import ... but I've seen it make some users get a bit careless.

We will learn Python through all three of these results. The preferred method for homework submission will *usually* be Option #3 (Jupyter notebooks) since they are easy for others to use.

For more details, see this nice discussion of the Python interpreter on the official site, here: https://docs.python.org/3/tutorial/interpreter.html

Python PATH for importing functions, installing programs

What if you are working in some directory and want to import and then use a Python function? It is usually good practice to keep all your code -- at least detailed packages -- in a place well separated from what you are working on.

In order for you to import and utilize that code, it has to be in your Python *path* . How do you know what your path is? Here's how:

```
from os import sys
print(sys.path)

#or, more elegantly & better formatted...
print('2')
print('\n'.join(sys.path))
print('3')
#stuff that you added to sys.path
import os
pypath=os.environ['PYTHONPATH'].split(os.pathsep)
print('\n'.join(pypath))
```

```
['/Users/thaynecurrie/Research/coding/Python/ScientificPythonNotes_2024/Basic', '/Users/thaynecurrie/Research/WFC/SOAPY', '/Users/thaynecurrie/Research/Github/repositories/charis-dpp_test', '/Users/thaynecurrie/Research/Github/repositories/charis-dpp_test/python', '/Users/thaynecurrie/Research/Models/RADMC3D/radmc3d-2.0/python', '/Users/thaynecurrie/Research/PythonPhot', '/Users/thaynecurrie/anaconda2/envs/py310/lib/python310.zip', '/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10', '/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/lib-dynload', '', '/Users/thaynecurrie/.local/lib/python3.10/site-packages', '/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-packages', '/Users/thaynecurrie/Research/Github/repositories/orvara_py310/orvara', '/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-packages/xaosim-2.0.0-py3.10.egg', '/Users/thaynecurrie/Research/Models/AggScatVIR/AggScatVIR/python', '/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-packages/pymcfost-0.0.0-py3.10.egg']
2
/Users/thaynecurrie/Research/coding/Python/ScientificPythonNotes_2024/Basic
/Users/thaynecurrie/Research/WFC/SOAPY
/Users/thaynecurrie/Research/Github/repositories/charis-dpp_test
/Users/thaynecurrie/Research/Github/repositories/charis-dpp_test/python
/Users/thaynecurrie/Research/Models/RADMC3D/radmc3d-2.0/python
/Users/thaynecurrie/Research/PythonPhot
/Users/thaynecurrie/anaconda2/envs/py310/lib/python310.zip
/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10
/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/lib-dynload

/Users/thaynecurrie/.local/lib/python3.10/site-packages
/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-packages
/Users/thaynecurrie/Research/Github/repositories/orvara_py310/orvara
/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-packages/xaosim-2.0.0-py3.10.egg
/Users/thaynecurrie/Research/Models/AggScatVIR/AggScatVIR/python
/Users/thaynecurrie/anaconda2/envs/py310/lib/python3.10/site-packages/pymcfost-0.0.0-py3.10.egg
3
/Users/thaynecurrie/Research/WFC/SOAPY/
/Users/thaynecurrie/Research/Github/repositories/charis-dpp_test/
/Users/thaynecurrie/Research/Github/repositories/charis-dpp_test/python/
/Users/thaynecurrie/Research/Models/RADMC3D/radmc3d-2.0/python/
/Users/thaynecurrie/Research/PythonPhot/
```

The `sys.path` variable prints out your entire list of paths available for Python to import functions. The `\n.join(sys.path` shows the same information with each path on a new

line.

The last print statement shows paths that are in your `$PYTHONPATH` environment variable. This is something that you have to set in your shell.

Now what if there's a Python package that you want to install? Well, you have a couple of options. The two most common installation methods are using `conda` -- if you have the Anaconda distribution of Python -- or `pip`. If the thing you are wanting to install is a very nice, official package (e.g. `numpy`), it should have its own installer script that you can use from `conda` or `pip`.

Examples

(installing or upgrading `numpy`) (note: you should only need to upgrade numpy to new versions: an installation comes standard with Anaconda)

```
conda install numpy
```

or

```
pip install numpy
```

Some caveats. First, `conda` is SLOW: it will carefully check for (what it thinks are) package incompatibilities. And if you are not careful, it may start overwriting packages you currently have to solve the incompatibility. Or conversely, `pip` might do an installation or upgrade that breaks packages a bit more easily.

For more details, see this nice discussion of the Python interpreter on the official site, here: https://docs.python.org/3/tutorial/interpreter.html

```
```