Monday, April 9, 2018

Unit 3: Introduction to Python

Reading:

Ch 8 in *PCfB*

Online resources (we'll start a discussion on Canvas)

Topics:

- what is Python?
- how to run Python
- numbers, strings, variables
- getting input data from the command line
- first program: temperature converter

What is Python?

Python is a **programming language**

- a notation for describing what we want a computer to do

Python is an “imperative” language

- we need to have an **algorithm**, a plan for solving a problem
- write statements in Python that tell the machine which steps to perform (“if this condition is true do that”, “repeat the following steps until another condition is true”, …)

History

The language was defined by computer scientist Guido van Rossum

- first released to the public in 1991
- has been free (open source) since the first version
- latest version is 3.6

The language is named after Monty Python's Flying Circus (British comedy group)

Python 2 vs Python 3

Version 3 of the language was released in 2008

- there are a lot of incompatibilities with Python 2, and many projects have been slow to adapt (including many scientific libraries)
- Python 2 is still supported (until 2020)

PCfB uses Python 2 😞

For the most part this won't be a problem but I'll have to point out differences as we encounter them…

How to Run Python

There are two ways to use Python, and both ways are similar to how we use `bash`:

a) start an interactive session, type Python commands in a REPL (read-execute-print loop)

b) put Python statements in a text file, tell Python to run the file

If you installed Python using Anaconda (our recommendation) then simply type `python` in a terminal window to run Python

Type this command to verify you have Python installed:

```
$ python --version
Python 3.6.3 :: Anaconda, Inc.
```

Mac users: *if you use the version installed by Apple, or download Python directly from python.org, start Python with this command:*

```
$ python3
```

Interactive Python

Just type `python` to start a new interactive session

- Python's prompt is three greater-than signs
- type ^D (control-D) to stop the session and return to the shell

The simplest type of Python statement is an **arithmetic expression**

- type an equation and hit the return key
- Python evaluates the expression and prints the result

Examples

```
>>> 6 * 7
42

>>> 2 * (3 + 4)
14

>>> from math import pi
>>> r = 2
>>> pi * r * r
12.566370614359172
```

Aside: IPython

Our instructions for installing Python recommended you install an application named **`ipython`**

- the “i” stands for “interactive”

If you want to run Python interactively it’s best to use ipython:

```
$ ipython
Python 3.6.3 |Anaconda, Inc.| (default, Dec  5 2017)
IPython 6.2.1 -- An enhanced Interactive Python. Type '?'
for help.

In [1]:
```

IPython displays its own prompt

IPython is a “wrapper” that adds some extra bells and whistles for interactive sessions

Notice how it runs the same version of Python

We’ll have more on IPython later in the term — for now it doesn’t matter which program you use in interactive sessions.

Python Programs

The other way to run Python is to put a series of Python statements in a plain text file

- by convention file names end with `.py`

There is one big difference from interactive Python:

- when Python is executing a program from a file it does not print the value of an expression

Instead we have to use a **print statement**

- write the word "print" followed by the value you want to see
- the value needs to be enclosed in parentheses

Examples

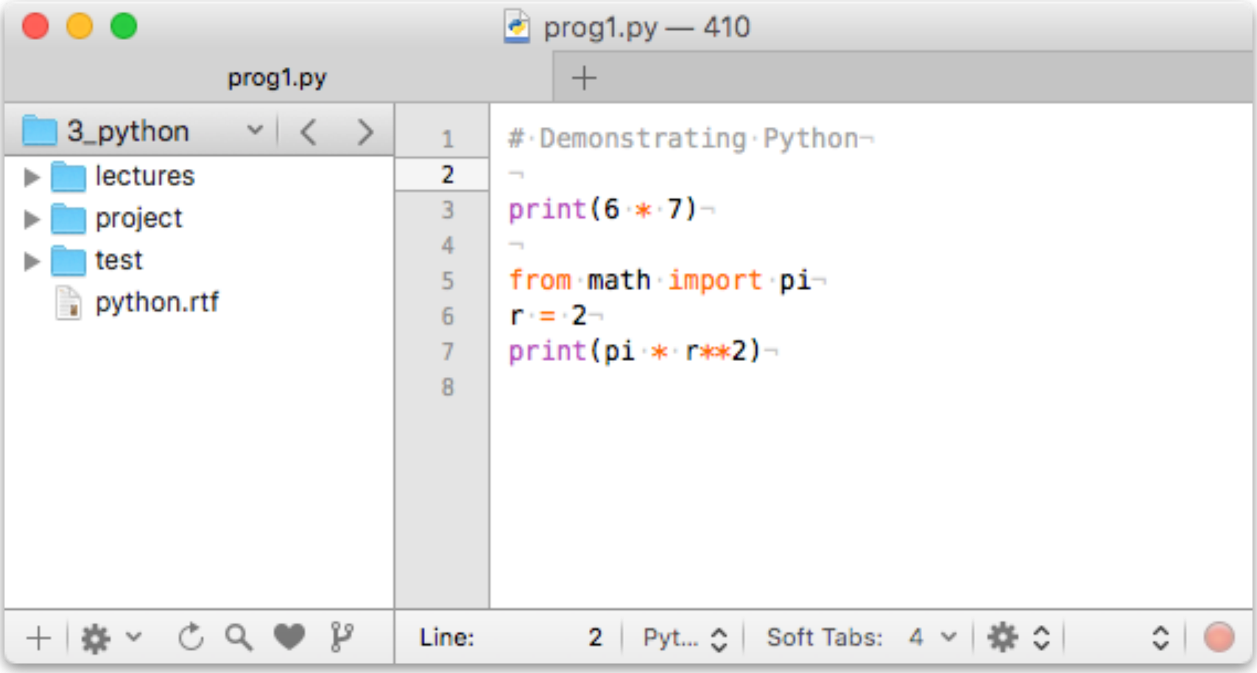
```
print(6 * 7)

print(pi * r**2)
```

2 vs 3: *Python2 does not use parentheses in print statements. You'll see examples like this in the book:*

```
print 6*7
print pi*r**2
```

Python Programs (cont’d)

Here is a file named `prog1.py`:



```
# Demonstrating Python

print(6 * 7)

from math import pi
r = 2
print(pi * r**2)
```

Line: 2

Pyt...

Soft Tabs: 4

To run the program start a terminal session, `cd` to the folder that contains the program, and tell Python to run it:

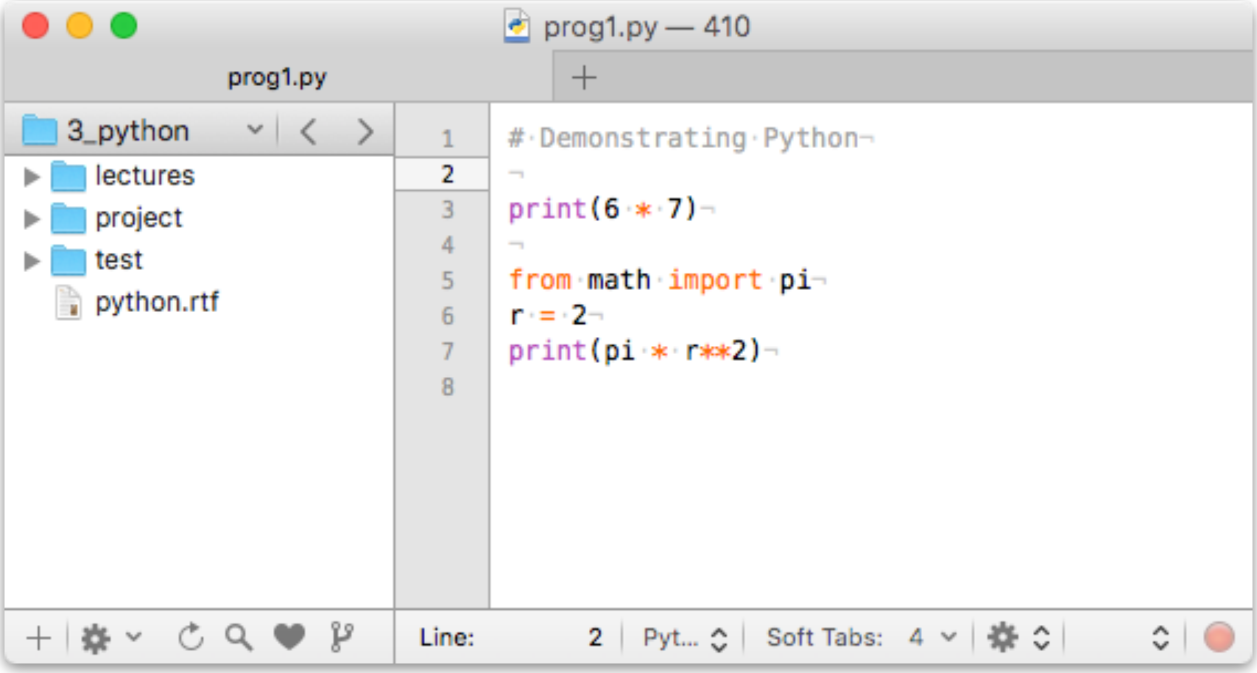
```
$ python prog1.py
42
12.5663706144
```

Aside: Syntax Coloring

In the screenshot on the previous page you probably noticed that words like `print`, `from`, and `import` were displayed with different colors

These colors are produced by the text editor — they are not part of the text file

- other programming editors (BBEdit, TextWrangler, Sublime Text, …) do something similar
- you can change the editor’s “theme” to choose a color style you like (or even make your own theme)



```
# Demonstrating Python

print(6 * 7)

from math import pi
r = 2
print(pi * r**2)

```

Line: 2

Soft Tabs: 4

Arithmetic Expressions

The examples above showed that Python can evaluate arithmetic expressions

- use a notation that is similar to what we see in math books
- the + and – keys stand for addition and subtraction
- use an asterisk `*` for multiplication
- use a slash `/` for division
- a double asterisk is used for exponentiation

Examples (from an `ipython` session):

```
In [1]: 6 * 7
Out[1]: 42

In [2]: 2 + 3 * 5 - 1
Out[2]: 16

In [3]: 5 ** 2
Out[3]: 25
```

Precedence Rules

The second example on the previous page shows how Python applies operators according to their **precedence**:

- exponentiation
- multiplication and division
- addition and subtraction

You can alter the order by using parentheses:

```
In [4]: 2 + 3 * 5 - 1
Out[4]: 16

In [5]: (2 + 3) * (5 - 1)
Out[5]: 20
```

Floats

Sometimes the result of an arithmetic operation is not an integer:

```
In [8]: 2 / 3
Out[8]: 0.6666666666666666
```

2 vs 3: *In Python2 the divide operator creates an integer:*

```
>>> 2 / 3
0
```

In computer science we say these are **floating point numbers** (aka "floats") instead of real numbers

- real numbers can have an infinite number of digits (*e.g.* ⅓ or π)
- numbers in a computer have to be truncated to a finite number of digits (0.3333333333333333 or 3.141592653589793)

The term "floating point" refers to the technique used to store real numbers in single "word" in memory (typically 64 bits per word)

Typing Floats

Type a decimal point if you want an expression to use floats:

```
In [9]: 6.0 * 7.0
Out[9]: 42.0
```

Combining Floats and Ints

An expression can have a combination of floating point numbers and integers (*aka* “ints”)

- Python will convert ints to floats to evaluate the expression
- the result will be a float

```
In [10]: 6 * 7.0
Out[10]: 42.0

In [11]: 2 / 3 * 27
Out[11]: 18.0
```

Variables

In Python a **variable** is a name attached to a value

Define a variable using an **assignment statement**

- write the variable name, an equal sign, and an expression
- Python evaluates the expression, then attaches the value to the variable

```
In [20]: x = 6 * 7
```

Note: in an interactive session there is no output from an assignment statement

After we define a variable we can use it in an expression:

```
In [21]: x * 2
Out[21]: 84
```

If you want to know the value of a variable just type its name — a name by itself is just a very simple expression with no operators:

```
In [22]: x
Out[22]: 42
```

Variable Names

The rules for variable names:

- must start with a letter or an underscore
- can be as long as you want and may contain digits
- upper and lower case are allowed (and are significant)

```
In [2]: alpha = 0.4

In [3]: year_1_total = 125

In [4]: A_long_name_is_allowed_but_is_hard_to_read = 0
```

Conventions

Your variable names should begin with a lower case letter

Choose a name that is mnemonic — you (or other people who read your program) should be able to understand what the variable is used for.

Part of a program's "style" points —
think about the names you choose,
make sure you use "best practice"

Variables Are Transient

When you quit an interactive session all your variables are discarded

- the next time you start an interactive session you'll begin with a "clean slate" — no variables will be defined

The same is true of running a program stored in a file

- when Python executes the first statement no variables are defined
- it's up to you to assign values (or maybe import them from a library)

Display a List of Defined Variables [`ipython`]

If you are using `ipython` you can type `who` or `whos` to see a list of names defined for the current session

```
In [8]: who
A_long_name_is_allowed_but_is_hard_to_read
alpha   x    year_1_total

In [9]: whos
Variable                                      Type     Info
-----------------------------------------------------------
A_long_name_is_allowed_but_is_hard_to_read    int       0
alpha                                         float     0.4
x                                             int       42
year_1_total                                  int       125
```

Variables Can Be Modified

In Python variables can change their values over time

- FYI: some languages also have symbols that keep the same value all the time (“constants”)

Example

```
In [1]: n = 5

In [2]: n * 10
Out[2]: 50
```

```
In [3]: n = 10
```

Assign a new value to `n`

```
In [4]: n * 10
Out[4]: 100
```

Same expression, different result

A Common Construct

We often talk about **updating** or **incrementing** a variable

- we want to assign a new value based on the current value

A very common construct (in almost all languages, not just Python):

```
x = x + 1
```

It seems illogical, but all it means is “evaluate x + 1 using the current value of x and then set x to the result.”

Another Assignment Operator

This construct is so common Python has a special assignment operator we can use:

```
x += y
```

means “update the value of x by adding y”

```
In [10]: x = 10

In [11]: x += 1

In [12]: x * 2
Out[12]: 22
```

Strings

Programs deal with text as well as numbers

- in Python a piece of text is called a **string**

When you define a string enclose the text in single or double quotes:

```
In [1]: name = 'Fred'

In [2]: prolog = "It was a dark and stormy night..."

In [3]: start_codon = "ATG"

In [4]: mood = '😀'

In [5]: uh_oh = 'I am \U0001F61F'
```

`\u` *or* `\U` *introduces a*
Unicode escape sequence

```
In [6]: whos

Variable      Type    Data/Info
-------------------------------
mood          str     😀
name          str     Fred
prolog        str     It was a dark and stormy night...
start_codon   str     ATG
uh_oh         str     I am 😟
```

Note how the value does not include the quotes — they're just used to delimit the text when you define the string

Our First Program

Let's use what we've seen so far to write our first Python program

We want to write a program that converts temperature from Fahrenheit to Celsius using the equation

$$C = \frac{5}{9} \times (F - 32)$$

- define a variable named `f` with the value we want to convert (e.g. 50)
- use arithmetic expressions to compute the result
- use a `print` statement to print the output message

Example (assuming `f` is set to 50):

```
$ python cels.py
50 F =  10.0 C
```

Sandbox

Before we start writing the program it's worth taking some time to make sure we can do the calculation in Python

Start an interactive session (either `python` or `ipython`) and define a variable named `f` with some temperature value

- choose a value where you know the answer, e.g. 77°F = 25°C

```
In [1]: f = 77
```

Now try typing some expressions to see how Python does the calculations.

- **Hint**: start with small parts, then combine those into the final expression

```
In [2]: f - 32
Out[2]: 45

In [3]: 5/9
Out[3]: 0.5555555555555556

In [4]: 5/9 * f - 32
Out[4]: 10.777777777777779
```

☠️ A bug! What did Python do here?

```
In [5]: 5/9 * (f - 32)
Out[5]: 25.0
```

This is right — subtract 32, then multiply by 5/9

Expert Tip: Use `ipython` for Interactive Experiments

What I showed on the previous page is an example of a very compelling reason for why Python is so popular.

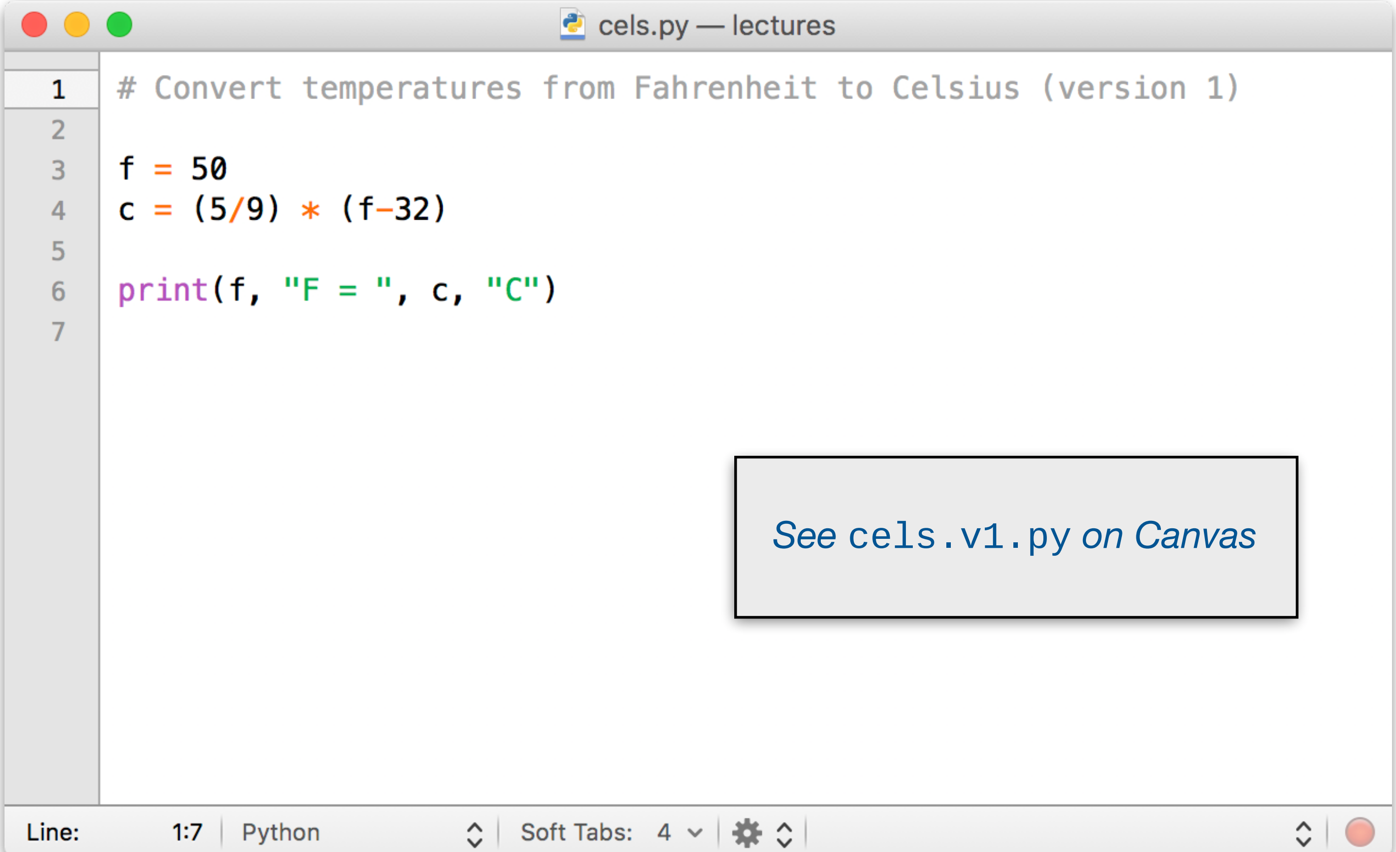
🔷 *Take advantage of Python's ability to work interactively by setting up "interactive experiments" to test code, then put it in your program.*

Once you have figured out what you need add it to your program, or even copy and paste from the interactive session into your program file.

> *If you ever find yourself asking "what happens if …" or "is … legal in Python?" set up an interactive experiment and try it out*

`cels.v1.py`

Here is the first version of the program — just assign values to `f` and c and print the results



```
# Convert temperatures from Fahrenheit to Celsius (version 1)

f = 50
c = (5/9) * (f-32)

print(f, "F = ", c, "C")
```

See `cels.v1.py` *on Canvas*

Notes:

`print` is a function (in Python 3)

pass it any number of items, Python will print them all, separated by spaces

this call is passed a float, a string, another float, and another string

Operations with Strings

We can write expressions where the data and variables are strings.

The **+** symbol tells Python to attach one string to another:

```
In [7]: name
Out[7]: 'Fred'

In [8]: fullname = 'Weasley' + ', ' + name

In [9]: fullname
Out[9]: 'Weasley, Fred’
```

The ***** symbol tells Python to repeat a string a specified number of times:

```
In [10]: s = 'no!'

In [11]: s * 3
Out[11]: 'no!no!no!'
```

Important concept: *The meaning of an* ***operator*** *(+, -, *, etc) depends on the* ***types of the operands*** *in the expression*

Substrings

If s refers to a string, the notation `s[i]` stands for "the character at location `i` in s”

- pronounced “s sub `i`”
- based on mathematical notation: s_i

Important: the first character is at location 0

```
In [13]: first = 'Hermione'

In [14]: last = 'Granger'

In [15]: first[0]
Out[15]: 'H'

In [16]: last[3]
Out[16]: 'n'

In [17]: first[0] + last[0]
Out[17]: 'HG'

In [18]: last + ', ' + first[0]
Out[18]: 'Granger, H'
```

Functions

An expression can also contain function names

- examples are `sin`, `cos`, `sqrt` (square root) from math
- there are also string functions like `len` (length) and `split` (used to break a string into smaller pieces)

Some terminology:

- when we use a function we say we **call** the function, or that an expression contains a function call
- the function is **passed** one or more arguments
- the arguments are written in parentheses after the name of the function
- functions **return** results which are used by the rest of the expression

Examples of Function Calls

(1) Pass a floating point number to `int`, it will return an integer (by truncating)

```
In [24]: int(3.75)
Out[24]: 3
```

(2) `round` is similar but "rounds off" to the nearest integer:

```
In [25]: round(3.75)
Out[25]: 4
```

(3) `pow` takes two arguments, n and m, and returns the value of n^m:

```
In [26]: pow(2,8)
Out[26]: 256
```

When a function has two or more arguments they are separated by a comma

(4) `len` returns the number of characters in a string (note that spaces and punctuation are counted as characters):

```
In [28]: fullname
Out[28]: 'Weasley, Fred'

In [29]: len(fullname)
Out[29]: 13
```

Functions Return Values

These examples emphasize the fact that when we call a function it returns a value that we can use just like other values:

```
In [1]: s = 'One fish, two fish, red fish, blue fish'

In [2]: x = 12

In [3]: y = 7

In [4]: n = len(s)

In [6]: z = max(x,y)

In [7]: whos

Variable   Type    Data/Info
----------------------------
n          int     39
s          str     One fish, two fish, red fish, ...
x          int     12
y          int     7
z          int     12


In [8]: len(s) * max(x,y)
Out[8]: 468
```

Cleaning Up `cels.py`

Let's use `round` to clean up the output of `cels.py`

This is what we want to see when we convert 70°F to Celsius:

```
$ python cels.py
70 F =  21.1 C
```

We can do this by calling `round` with two arguments. If we pass a second argument it used as the number of digits after the decimal place:

```
In [1]: c = (5/9) * (70-32)

In [2]: c
Out[2]: 21.11111111111111

In [3]: round(c)
Out[3]: 21

In [4]: round(c,1)
Out[4]: 21.1

In [5]: round(c,4)
Out[5]: 21.1111
```

Note again that I'm using an interactive session to make sure I know how round works before I put it in my program…

***Challenge**: Download `cels.v1.py`, modify it so the output temperature is printed with 1 decimal point*

Libraries

The functions used in examples so far (`round`, `len`, `max`, *etc*) are **built-in functions**

Other functions need to be **imported** from a **library** using an import statement

```
In [1]: sqrt(9)
NameError: name 'sqrt' is not defined

In [2]: from math import sqrt

In [3]: sqrt(9)
Out[3]: 3.0

In [4]: from random import randint, normalvariate

In [5]: randint(1,10)
Out[5]: 2

In [6]: normalvariate(100, 15)
Out[6]: 108.43290270761828
```

Some libraries also have useful variables as well as functions:

```
In [10]: from math import pi, cos

In [11]: pi
Out[11]: 3.141592653589793

In [12]: cos(2*pi)
Out[12]: 1.0
```

Libraries (cont'd)

When you installed Python you got several dozen libraries

For the projects in this class if you need a library function we'll give you all the information you need

- the name of the library
- the name of the function
- examples of how to call the function

You are welcome to browse the online documentation to see if there are other functions you might find useful

There are also tons of "third-party" libraries. Some that you got when you installed conda and that we'll use later in the term are:

`pandas`	functions for creating and using "data frames" (2D tables similar to spreadsheet tables)
`numpy, scipy`	numerical processing, including statistics
`matplotlib`	functions for data visualization, comparable to R

Command Line Arguments

The next topic is going to help us make more useful programs

The idea is to be able to get **values from the command line** when we run a Python program

For example, it would be nice if we could specify the value of the temperature to convert when we run `cels.py` instead of having to edit the program each time

```
$ python cels.py 50
50.0 F =  10.0 C

$ python cels.py 70
70.0 F =  21.1 C

$ python cels.py 100
100.0 F =  37.8 C

$ python cels.py 32
32.0 F =  0.0 C

$ python cels.py 0
0.0 F =  -17.8 C
```

`argv`

To get values from the command line we need a variable named **`argv`** that is defined in the `sys` library

- "argv" is short for "argument vector"
- in Python it's a **list** — we'll learn more about lists next week, but for now we just need to know how to access the command line arguments

Add this import statement to the front of the program:

```
from sys import argv
```

Now we can use the notation `argv[i]` to refer to "command line argument number i"

- if a program wants to access the command line arguments it refers to them as `argv[1]`, `argv[2]`, *etc*

`cels.py` Command Line

Our `cels.py` program will have one argument

- the expression `argv[1]` refers to the Fahrenheit temperature specified by the user on the command line

`argv` is a List of Strings

An important detail: **all the items in `argv` are strings**

When we start the program with

```
$ cels.py 50
```

the OS stores the string `'50'` in `argv[1]`

Know the Difference Between a Number and a String of Digits

- a number is an abstract quantity from math
- a string is a sequence of characters; we use a string of digits to refer to the name of a number

```
In [1]: s = '50'

In [2]: t = 50

In [3]: whos
Variable   Type    Data/Info
----------------------------
s          str     50
t          int     50
```

Type Conversion

For `cels.py` to use the command line argument as a temperature value it has to convert it to a float

In Python the names of types — int, float, str — are also the names of functions

call `int(x)` to convert x into an integer

call `float(x)` to convert x to a float

call `str(x)` to create a string from the name of x

```
In [4]: int(13.7)
Out[4]: 13

In [5]: float(13)
Out[5]: 13.0

In [6]: str(13)
Out[6]: '13'

In [7]: float('50')
Out[7]: 50.0

In [8]: int('50')
Out[8]: 50
```

Final Version: `cels.py`

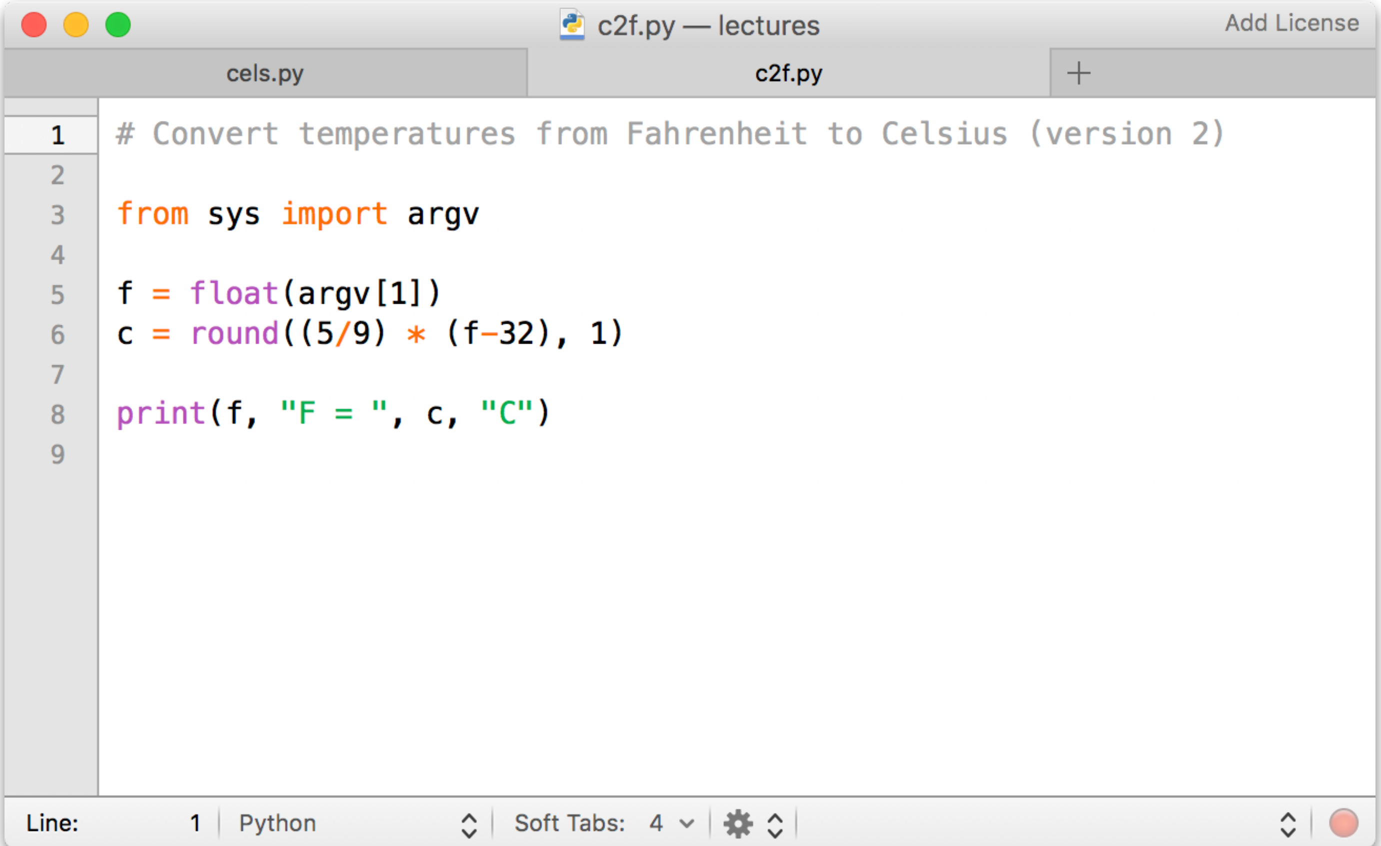
Here is the new version of the Celsius program (see `cels.py` on Canvas)

On line 5 replace

```
f = 50
```

with

```
f = float(argv[1])
```



```
1  # Convert temperatures from Fahrenheit to Celsius (version 2)
2
3  from sys import argv
4
5  f = float(argv[1])
6  c = round((5/9) * (f-32), 1)
7
8  print(f, "F = ", c, "C")
9
```

Line: 1 | Python | Soft Tabs: 4

Style Notes

We didn’t have to save the value of `argv[1]` in the variable `f`

For that matter we didn’t have to save the result of the expression in c

This program could be written as a simple “one-liner”:

```
print(round((5/9) * (float(argv[1])-32), 1))
```

(1) Assignments Simplify Expressions

Creating variables `f` and `c` makes the print statement much easier to read

This is generally a matter of taste, and the tradeoff is adding extra lines and figuring out good names for the variables.

(2) Avoid `argv` in Expressions

This rule is more strict: *avoid references to* `argv` *in the main part of the program*

Always start by saving values from `argv` in variables