

# Introduction to Python

## Part 1

Brian Gregor

Research Computing Services  
Information Services & Technology



# RCS Team and Expertise

- Our Team
  - Scientific Programmers
  - Systems Administrators
  - Graphics/Visualization Specialists
  - Account/Project Managers
  - Special Initiatives (Grants)
- Maintains and administers the Shared Computing Cluster
  - Located in Holyoke, MA
  - ~17,000 CPUs running Linux
- Consulting Focus:
  - Bioinformatics
  - Data Analysis / Statistics
  - Molecular modeling
  - Geographic Information Systems
  - Scientific / Engineering Simulation
  - Visualization
- CONTACT US: [help@scv.bu.edu](mailto:help@scv.bu.edu)

# About You

- Working with Python already?
- Have you used any other programming languages?
- Why do you want to learn Python?

# Running Python for the Tutorial

- If you have an SCC account, log into it and use Python there.
  - Run:

```
module load python/3.6.2  
spyder &
```

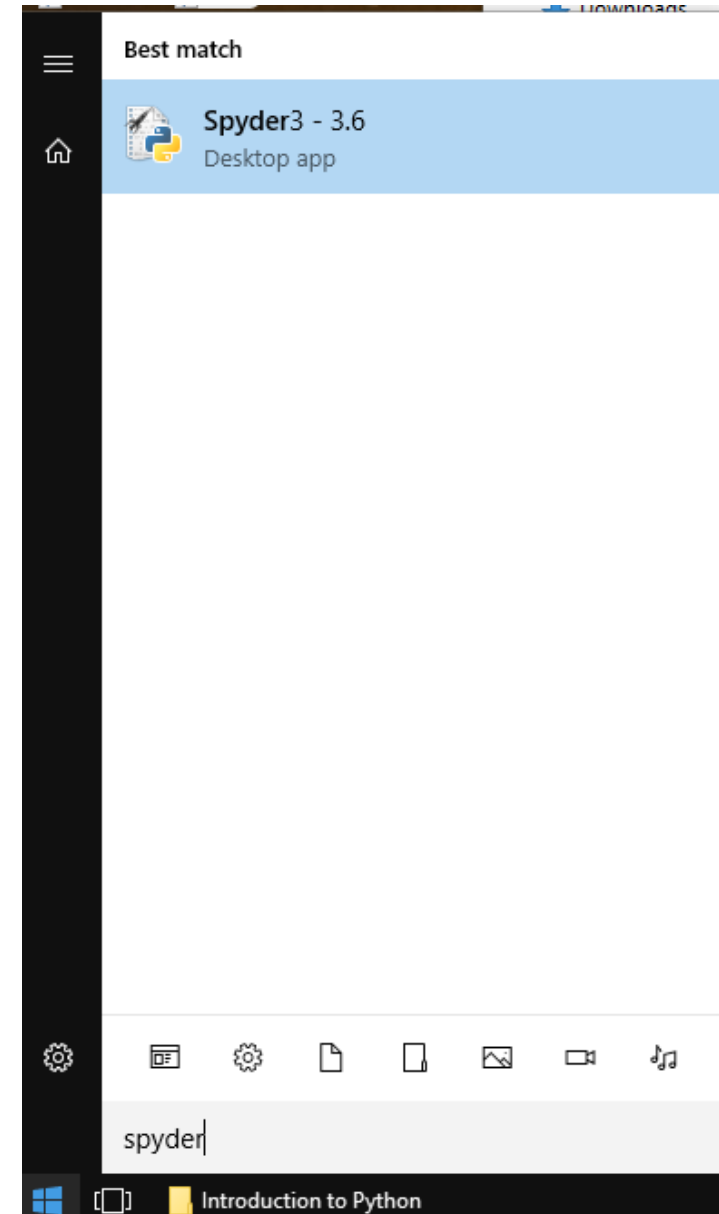
# Links on the Rm 107 Terminals

- On the Desktop open the folders:  
Tutorial Files → RCS\_Tutorials → Tutorial Files → Introduction to Python
- Copy the whole *Introduction to Python* folder to the desktop or to a flash drive.
  - When you log out the desktop copy will be deleted!

■

# Run Spyder

- Click on the Start Menu in the bottom left corner and type: `spyder`
- After a second or two it will be found. Click to run it.



# Running Python: Installing it yourself

- There are **many** ways to install Python on your laptop/PC/etc.
- <https://www.python.org/downloads/>
- <https://www.anaconda.com/download/>
- <https://www.enthought.com/product/enthought-python-distribution/>
- <https://python-xy.github.io/>



# BU's most popular option: Anaconda

- <https://www.anaconda.com/download/>
- Anaconda is a packaged set of programs including the Python language, a huge number of libraries, and several tools.
- These include the **Spyder** development environment and Jupyter notebooks.
- Anaconda can be used on the SCC, with some caveats.



# Python 2 vs. 3

- Python 2: released in 2000, Python 3 released in 2008
  - Python 2 is in “maintenance mode” – no new features are expected
- Py3 is not completely compatible with Py2
  - For learning Python these differences are almost negligible
- Which one to learn?
  - If your research group / advisor / boss / friends all use one version that's probably the best one for you to choose.
  - If you have a compelling reason to focus on one vs the other
  - Otherwise just choose Py3. This is where the language development is happening!

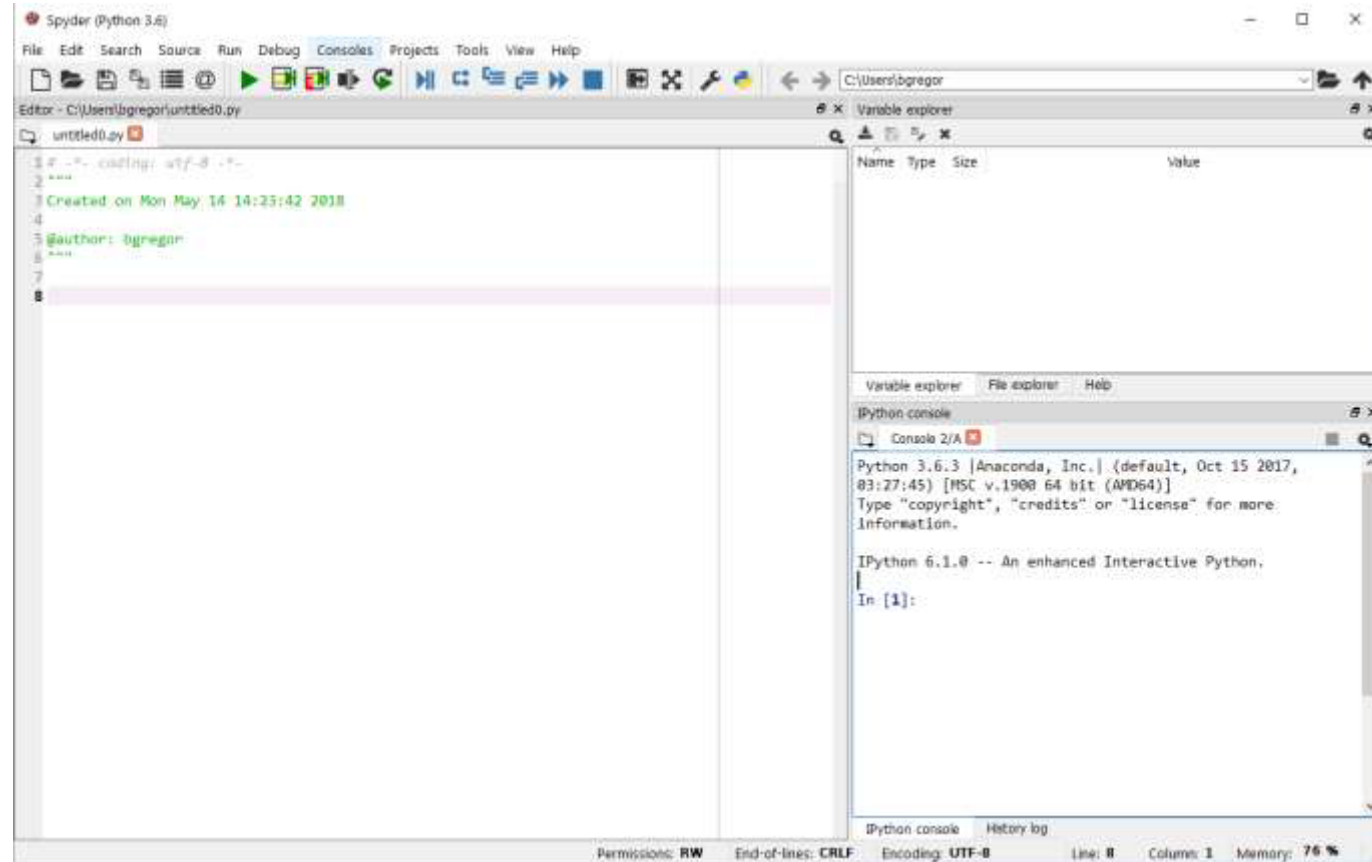
# Spyder – a Python development environment

- Pros:

- Faster development
- Easier debugging!
- Helps organize code
- Increased efficiency

- Cons

- Learning curve
- Can add complexity to smaller problems



# Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# Tutorial Outline – Part 2

- Functions
- Tuples and dictionaries
- Modules
- numpy and matplotlib modules
- Script setup
- Classes
- Debugging

# Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# What is Python?

- Python...
  - ...is a general purpose **interpreted** programming language.
  - ...is a language that supports multiple approaches to software design, principally **structured** and **object-oriented** programming.
  - ...provides **automatic memory management** and **garbage collection**
  - ...is **extensible**
  - ...is **dynamically** typed.
- By the end of the tutorial you will understand all of these terms!

# Some History

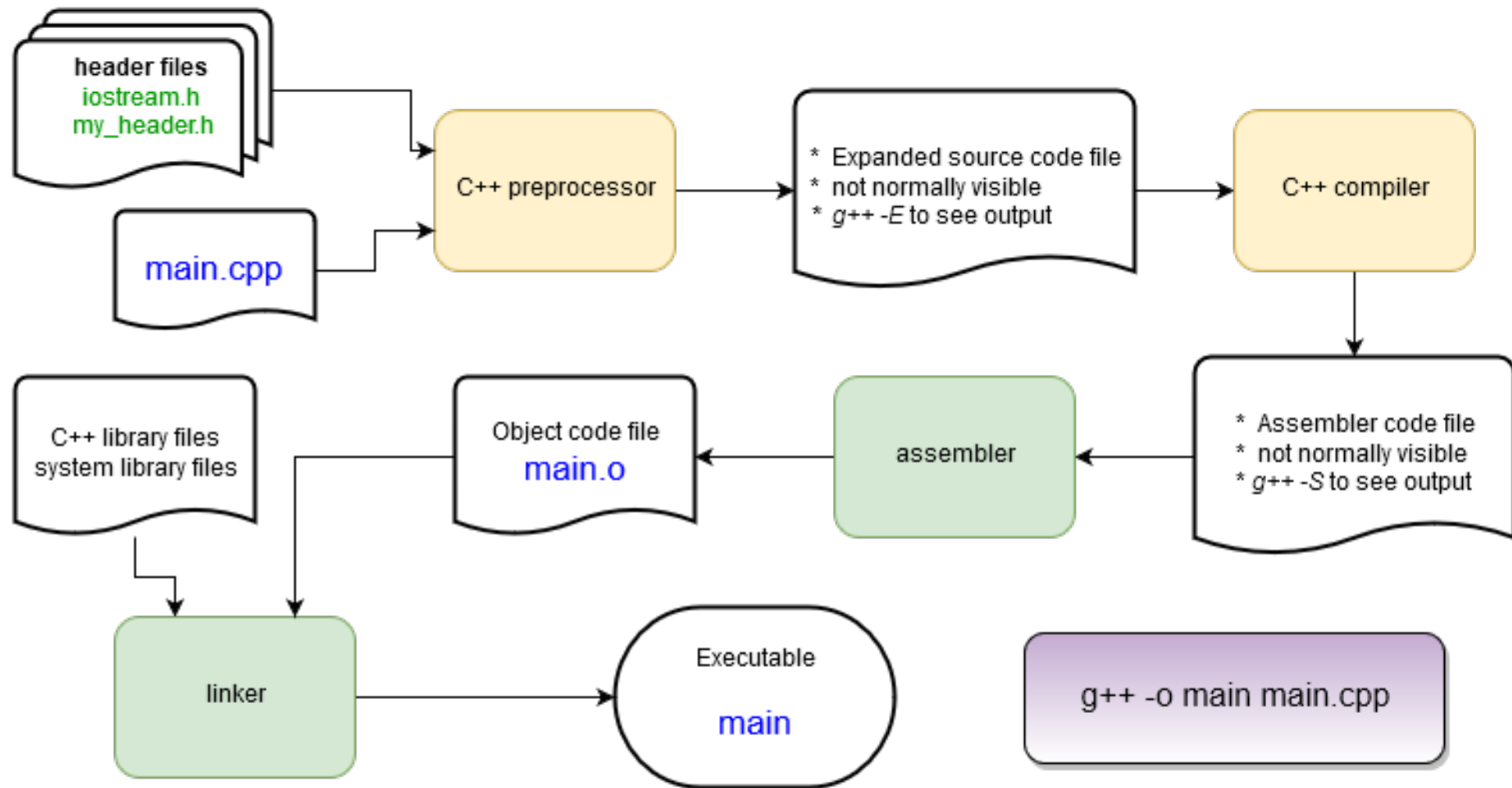
- “Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas...I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).”

–Python creator Guido Van Rossum, from the foreword to *Programming Python* (1<sup>st</sup> ed.)

- Goals:
  - An easy and intuitive language just as powerful as major competitors
  - Open source, so anyone can contribute to its development
  - Code that is as understandable as plain English
  - Suitability for everyday tasks, allowing for short development times

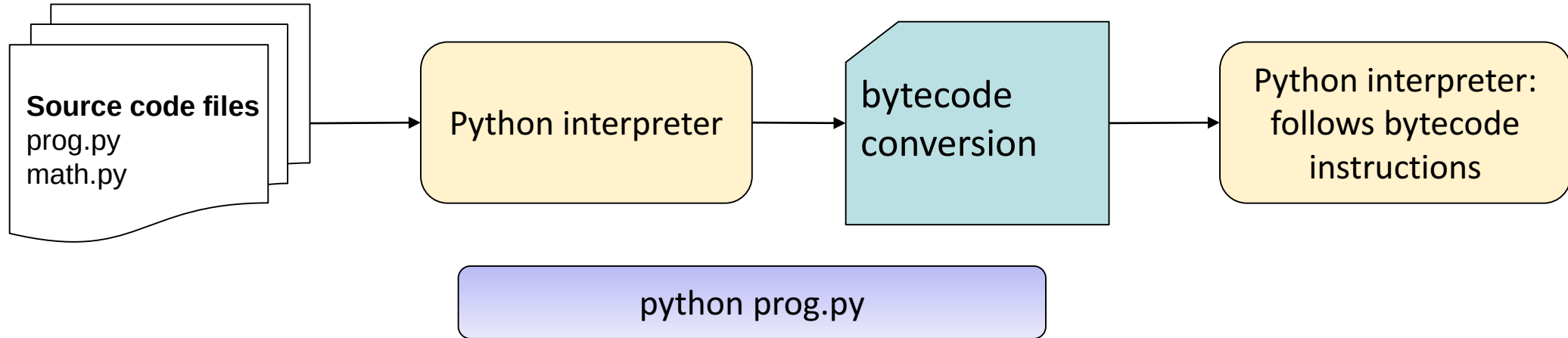


# Compiled Languages (ex. C++ or Fortran)





# Interpreted Languages (ex. Python or R)



- Clearly, a lot less work is done to get a program to start running compared with compiled languages!
- Bytecodes are an internal representation of the text program that can be efficiently run by the Python interpreter.
- The interpreter itself is written in C and is a compiled program.

# Comparison

## Interpreted

- Faster development
- Easier debugging
  - Debugging can stop anywhere, swap in new code, more control over state of program
- (almost always) takes less code to get things done
- Slower programs
  - Sometimes as fast as compiled, rarely faster
- Less control over program behavior

## Compiled

- Longer development
  - Edit / compile / test cycle is longer!
- Harder to debug
  - Usually requires a special compilation
- (almost always) takes more code to get things done
- Faster
  - Compiled code runs directly on CPU
  - Can communicate directly with hardware
- More control over program behavior

# The Python Prompt

- The standard Python prompt looks like this:

```
[bgregor@scc2 bg]$ python
Python 3.6.2 (default, Aug 30 2017, 15:46:55)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- The IPython prompt in Spyder looks like this:

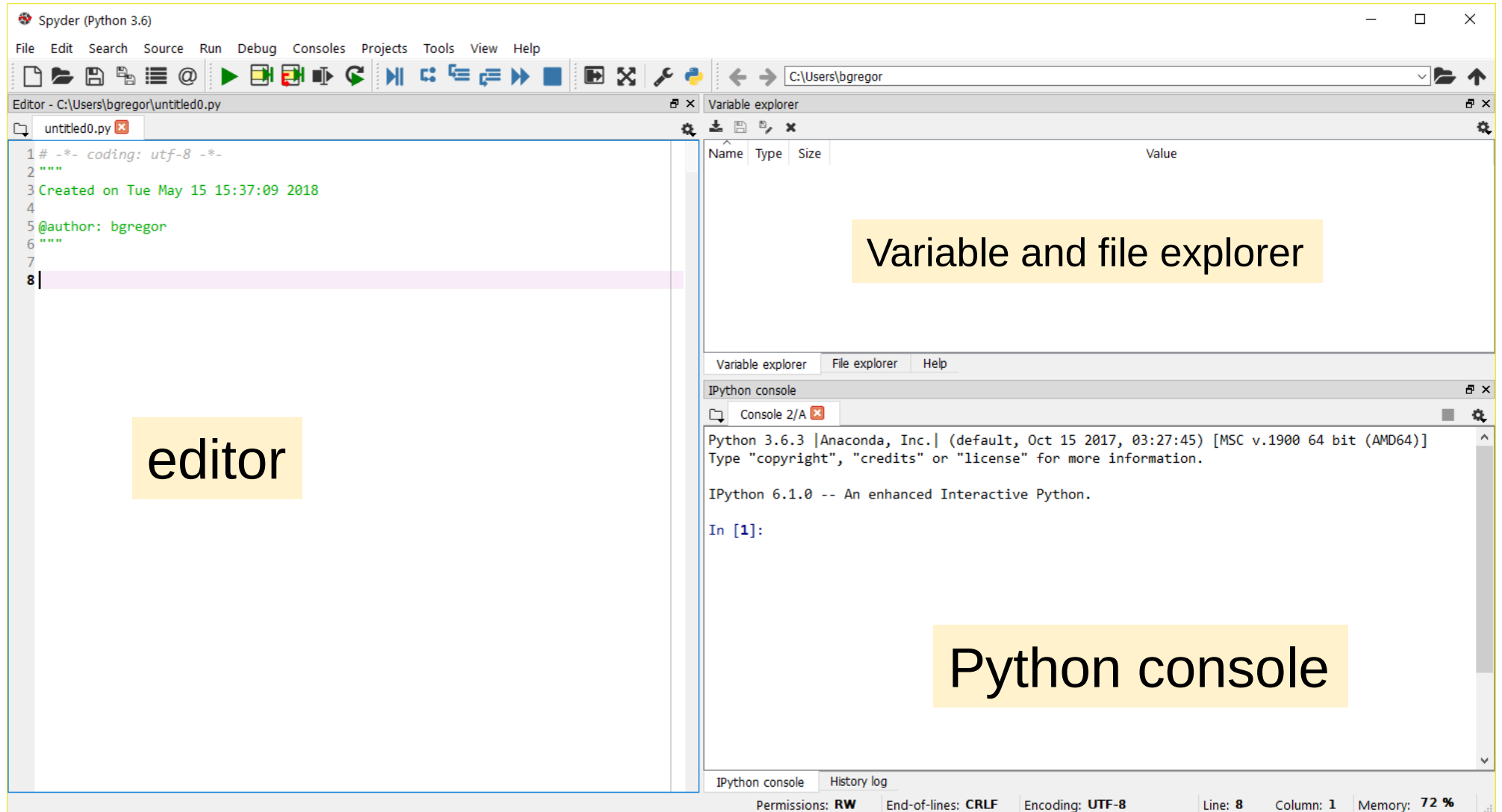
```
Python 3.6.3 [Anaconda, Inc.] (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]:
```

- IPython adds some handy behavior around the standard Python prompt.

# The Spyder IDE



# Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# Operators

- Python supports a wide variety of operators which act like functions, i.e. they do something and return a value:

- Arithmetic:      +      -      \*      /      %      \*\*
- Logical:          and      or      not
- Comparison:      >          <          >=          <=          !=          ==
- Assignment:      =
- Bitwise:          &          |          ~          ^          >>          <<
- Identity:          is          is not
- Membership:      in          not in

# Try Python as a calculator

- Go to the Python prompt.
- Try out some arithmetic operators:

+      -      \*      /      %      \*\*      ==      (   )

- Can you identify what they all do?

```
Python 3.6.3 |Anaconda, Inc.| (default, Oct 15 2017, 03:27:45)  
Type "copyright", "credits" or "license" for more information.
```

```
IPython 6.1.0 -- An enhanced Interactive Python.
```

```
In [1]: 1 + 3  
Out[1]: 4
```

```
In [2]: 4*2  
Out[2]: 8
```

```
In [3]: |
```

# Try Python as a calculator

- Go to the Python prompt.
- Try out some arithmetic operators:

+      -      \*      /      %      \*\*      ==      ()

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division (Note: 3 / 4 is 0.75!)
%	Remainder (aka <i>modulus</i> )
**	Exponentiation
==	Equals



# More Operators

- Try some comparisons and Boolean operators. *True* and *False* are the keywords indicating those values:

```
In [15]: 4 > 5  
Out[15]: False
```

```
In [16]: 6 > 3 and 3 > 0  
Out[16]: True
```

```
In [17]: not False  
Out[17]: True
```

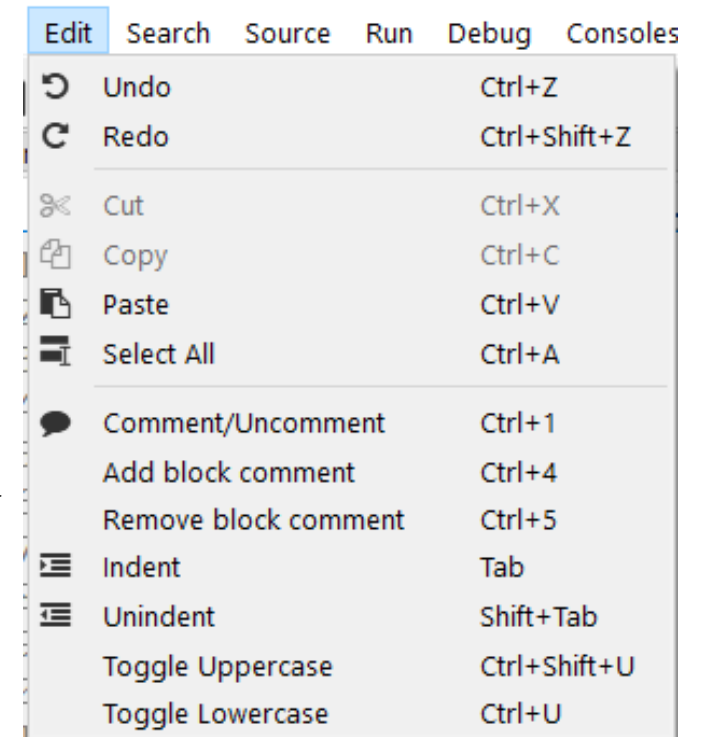
```
In [18]: True and (False or not False)  
Out[18]: True
```

```
In [19]:
```

# Comments

- # is the Python comment character. On any line everything after the # character is ignored by Python.
- There is no multi-line comment character as in C or C++.
- An editor like Spyder makes it very easy to comment blocks of code or vice-versa. Check the *Edit* menu

```
a=1
b=2
# this is a comment
c=3 # this is also a comment
# this is a
# multiline comment
```



# Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# Variables

- Variables are assigned values using the = operator
- In the Python console, typing the name of a variable prints its value
  - Not true in a script!
- Variables can be reassigned at any time
- Variable type is not specified
- Types can be changed with a reassignment

```
In [1]: a=1
```

```
In [2]: b=2
```

```
In [3]: a  
Out[3]: 1
```

```
In [4]: b  
Out[4]: 2
```

```
In [5]: a=b
```

```
In [6]: a  
Out[6]: 2
```

```
In [7]: b=-0.15
```

# Variables cont'd

- Variables refer to a value stored in memory and are created when first assigned
- Variable names:
  - Must begin with a letter (a - z, A - B) or underscore \_
  - Other characters can be letters, numbers or \_
  - Are case sensitive: capitalization counts!
  - Can be any reasonable length
- Assignment can be done *en masse*:

`x = y = z = 1`

- Multiple assignments can be done on one line:

`x, y, z = 1, 2.39, 'cat'`

Try these out!



# Variable Data Types

- Python determines data types for variables based on the context
- The type is identified when the program **runs**, called **dynamic typing**
  - Compare with compiled languages like C++ or Fortran, where types are identified by the programmer and by the compiler **before** the program is run.
- Run-time typing is very convenient and helps with rapid code development...but requires the programmer to do more code testing for reliability.
  - The larger the program, the more significant the burden this is!!

# Variable Data Types

- Available basic types:
  - Numbers: Integers and floating point (64-bit)
  - Complex numbers: `x = complex(3, 1)` or `x = 3+1j`
  - Strings, using double or single quotes: `"cat"` `'dog'`
  - Boolean: `True` and `False`
  - Lists, dictionaries, and tuples
    - These hold collections of variables
  - Specialty types: files, network connections, objects
- Custom types can be defined. This will be covered in Part 2.

# Variable modifying operators

- Some additional arithmetic operators that modify variable values:

Operator	Effect	Equivalent to...
$x += y$	Add the value of $y$ to $x$	$x = x + y$
$x -= y$	Subtract the value of $y$ from $x$	$x = x - y$
$x *= y$	Multiply the value of $x$ by $y$	$x = x * y$
$x /= y$	Divide the value of $x$ by $y$	$x = x / y$

- The  $+=$  operator is by far the most commonly used of these!



# Check a type

- A built-in function, *type()*, returns the type of the data assigned to a variable.
  - It's unusual to need to use this in a program, but it's available if you need it!
- Try this out in Python – do some assignments and reassignments and see what *type()* returns.

```
In [1]: a=1.0

In [2]: b=3

In [3]: c='Hello!'

In [4]: type(a)
Out[4]: float

In [5]: type(b)
Out[5]: int


In [6]: type(c)
Out[6]: str
```

# Strings

- Strings are a basic data type in Python.
- Indicated using pairs of single " or double "" quotes.
- Multiline strings use a triple set of quotes (single or double) to start and end them.
- Strings have many built-in functions...

```
'cat'  
"dog"  
"What's that?"  
'They said "hello"'  
''' This is  
    a multiline  
    string '''
```

# String functions

- In the Python console, create a string variable called *mystr*
- type: *dir(mystr)*
- Try out some functions: 
- Need help? Try:  
*help(mystr.title)*

```
len(mystr)
mystr.upper()
mystr.title()
mystr.isdecimal()
help(mystr.isdecimal)
```

# The len() function

- The len() function is not a string specific function.
- It'll return the length of any Python variable that contains some sort of countable thing.
- In the case of strings it is the number of characters in the string.

# String operators

- Try using the + and += operators with strings in the Python console.
- + concatenates strings.
- += appends strings.
- Index strings using square brackets, starting at 0.

```
a="Hello BU!"  
print(a[4])
```

# String operators

- Changing elements of a string by an index is **not allowed**:

```
In [79]: a='Hello BU!'

In [80]: a[4] = '0'
Traceback (most recent call last):

  File "<ipython-input-80-7c5733c2cb67>", line 1, in <module>
    a[4] = '0'

TypeError: 'str' object does not support item assignment
```

- Python strings are **immutable**, i.e. they can't be changed.

# String Substitutions

- Python provides an easy way to stick variable values into strings called *substitutions*

- Syntax for one variable:

```
'string with a %s' % variable
```



- For more than one:

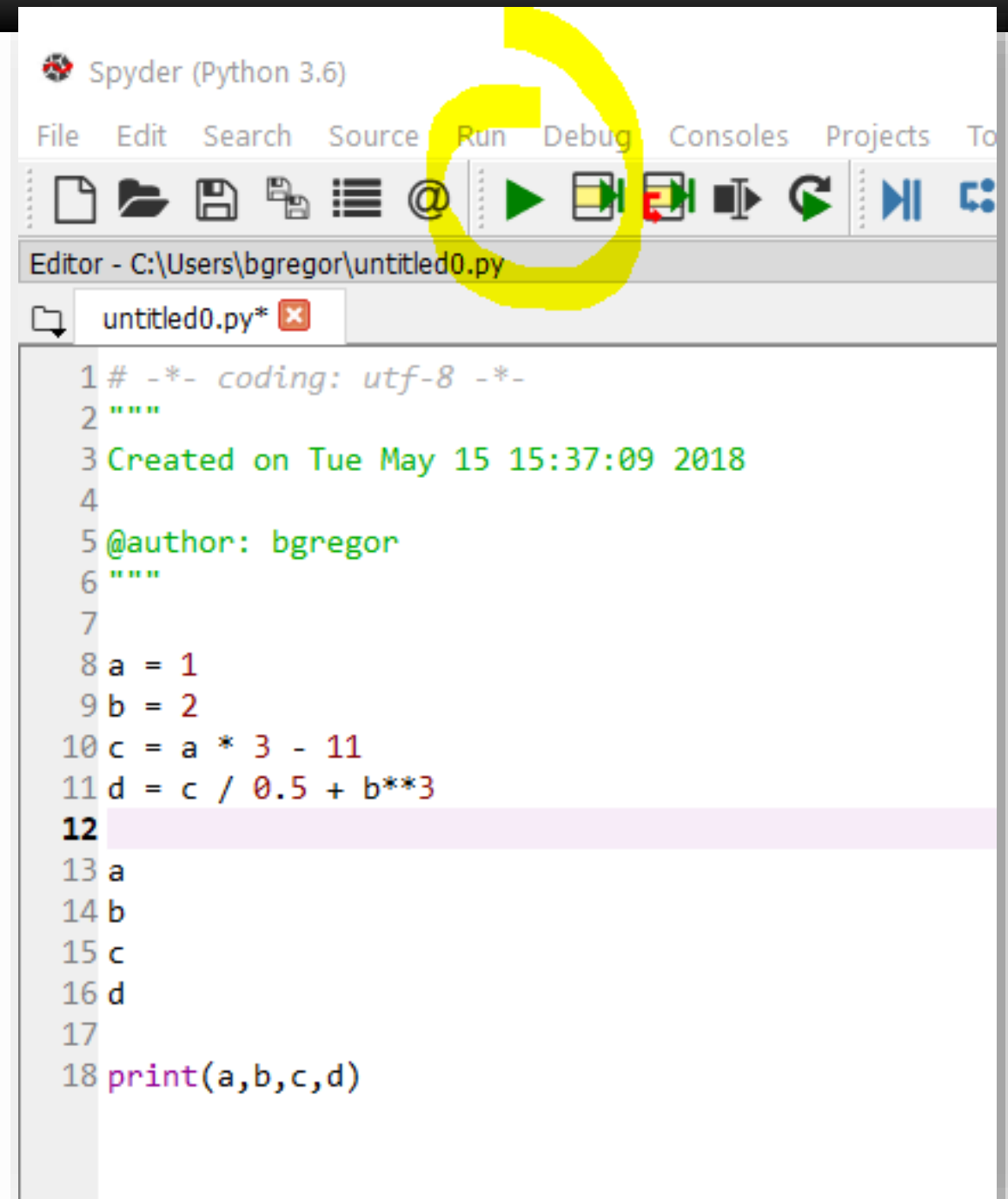
```
'x: %s  y: %s  z: %s' % (xval,yval,zval)
```



Variables are listed in the substitution order inside ()

# Variables with operators

- Operators can be combined freely with variables and variable assignment.
- Try some out again!
- This time type them into the editor. Click the green triangle to run the file. Save the file and it will run.



Spyder (Python 3.6)

File Edit Search Source Run Debug Consoles Projects To

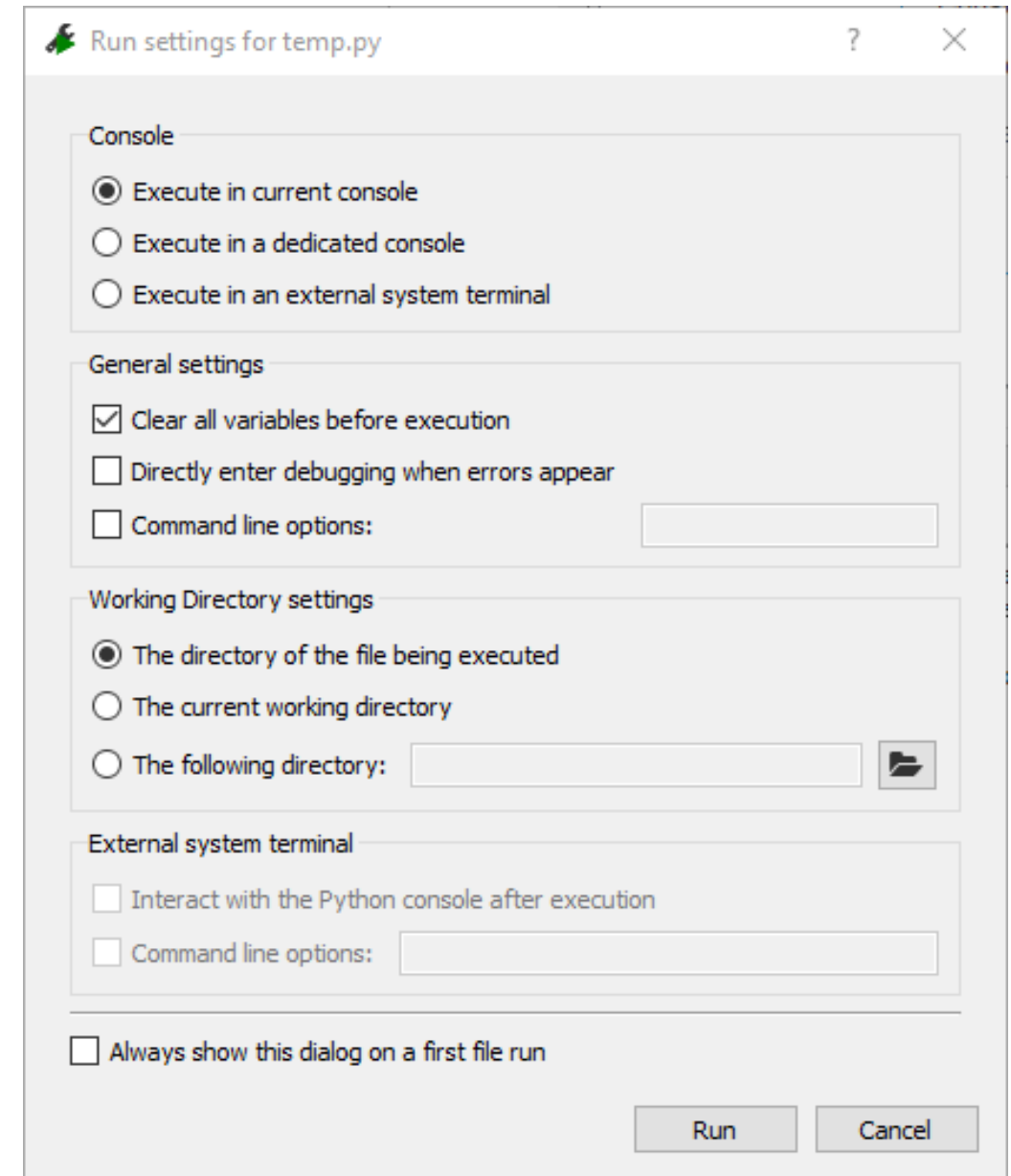
Editor - C:\Users\bgregor\untitled0.py

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue May 15 15:37:09 2018
4
5 @author: bgregor
6 """
7
8 a = 1
9 b = 2
10 c = a * 3 - 11
11 d = c / 0.5 + b**3
12
13 a
14 b
15 c
16 d
17
18 print(a,b,c,d)
```



# Spyder setup

- The first time you run a script Spyder will prompt you with a setup dialog:
- Just click “Run” to run the script. This will only appear once.



- The Variable Explorer window is displaying variables and types defined in the console.
- Only the *print* function printed values from the script.
  - Key difference between scripts and the console!

The screenshot shows the Spyder Python IDE interface. The main editor window displays a Python script named `untitled0.py` with the following content:

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue May 15 15:37:09 2018
4
5 @author: bgregor
6 """
7
8 a = 1
9 b = 2
10 c = a * 3 - 11
11 d = c / 0.5 + b**3
12
13 a
14 b
15 c
16 d
17
18 print(a,b,c,d)
```

The Variable Explorer window on the right displays the variables defined in the script:

Name	Type	Size	Value
a	int	1	1
b	int	1	2
c	int	1	-8
d	float	1	-8.0
x	int	1	1
y	int	1	1

The IPython console at the bottom shows the output of the script:

```
In [18]: runfile('C:/temp/untitled0.py', wdir='C:/temp')
1 2 -8 -8.0

In [19]:
```

Arrows indicate the flow of data: one arrow points from the Variable Explorer to the first bullet point, and another points from the `print(a,b,c,d)` line in the script to the console output.

# Tutorial Outline – Part 1

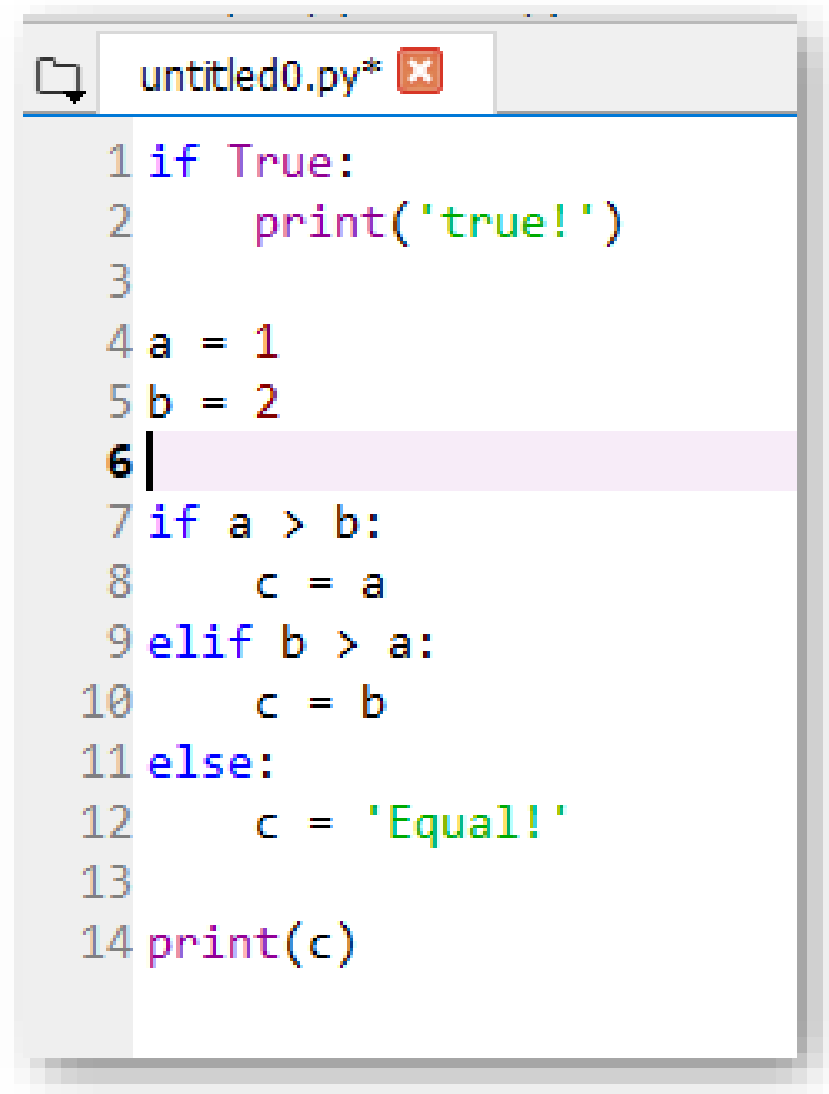
- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# If / Else

- *If*, *elif*, and *else* statements are used to implement conditional program behavior
- Syntax:

```
if Boolean_value:  
    ...some code  
elif Boolean_value:  
    ...some other code  
else:  
    ...more code
```
- *elif* and *else* are not required – used to chain together multiple conditional statements or provide a default case.

- Try out something like this in the Spyder editor.
- Do you get any error messages in the console?
- Try using an *elif* or *else* statement by itself without a preceding *if*. What error message comes up?



```
untitled0.py*  
1 if True:  
2     print('true!')  
3  
4 a = 1  
5 b = 2  
6  
7 if a > b:  
8     c = a  
9 elif b > a:  
10    c = b  
11 else:  
12    c = 'Equal!'  
13  
14 print(c)
```

# Indentation of code...easier on the eyes!

- C:

```
int x ;  
if (3 > 4) {  
x = 5 ;  
} else {  
x = 6 ;  
}
```

or

```
int x ;  
if (3 > 4) {  
    x = 5 ;  
} else {  
    x = 6 ;  
}
```

- Matlab:

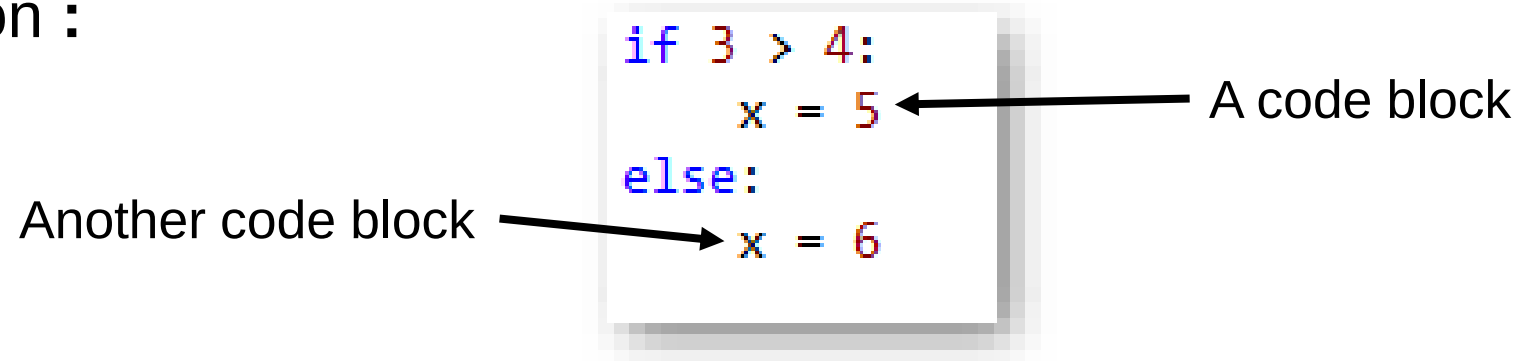
```
if (3 > 4)  
x = 5  
else  
x = 6  
end
```

or

```
if (3 > 4)  
    x = 5  
else  
    x = 6  
end
```

# The Use of Indentation

- Python uses whitespace (spaces or tabs) to define *code blocks*.
- Code blocks are logical groupings of commands. They are **always** preceded by a colon :



```
if 3 > 4:
    x = 5
else:
    x = 6
```

A code block


Another code block

- This is due to an emphasis on code readability.
  - Fewer characters to type and easier on the eyes!
- Spaces or tabs can be mixed in a file but **not** within a code block.

# If / Else code blocks

- Python knows a code block has ended when the indentation is removed.
- Code blocks can be nested inside others therefore *if-elif-else* statements can be freely nested within others.

```
a = 1
b = 2
if a <= b:
    c = a
    print('a <= b')
    if c == 1:
        print('c is 1')
print('out of the if statement')
```



- Note the lack of “end if”, “end”, curly braces, etc.



# File vs. Console Code Blocks

- Python knows a code block has ended when the indentation is removed.
- EXCEPT when typing code into the Python console. There an empty line indicates the end of a code block.
- Let's try this out in Spyder
- This sometimes causes problems when pasting code into the console.
- This issue is something the IPython console helps with.

# Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# Lists

- A Python list is a general purpose 1-dimensional container for variables.
  - i.e. it is a row, column, or vector of things
- Lots of things in Python act like lists or use list-style notation.
- Variables in a list can be of any type at any location, including other lists.
- Lists can change in size: elements can be added or removed
- **Lists are not meant for high performance numerical computing!**
  - We'll cover a library for that in Part 2
  - *Please* don't implement your own linear algebra with Python lists unless it's for your own educational interests.

# Making a list and checking it twice...

- Make a list with [ ] brackets.
- Append with the *append()* function
- Create a list with some initial elements
- Create a list with N repeated elements

Try these out yourself!  
Edit the file in Spyder and run it.  
Add some print() calls to see the lists.

```
list_1 = []  
  
list_1.append(1)  
list_1.append('A string!')  
list_1.append([])  
  
list_2 = [4, 5, -23.0+4.1j, 'cat']  
  
list_3 = 10 * [42]
```

# List functions

- Try `dir(list_1)`
- Like strings, lists have a number of built-in functions
- Let's try out a few...
- Also try the `len()` function to see how many things are in the list: `len(list_1)`

```
'append',  
'clear',  
'copy',  
'count',  
'extend',  
'index',  
'insert',  
'pop',  
'remove',  
'reverse',  
'sort']
```

# Accessing List Elements

- Lists are accessed by index.
  - All of this applies to accessing strings by index as well!
- Index #'s start at 0.
- List: `x = [ 'a', 'b', 'c', 'd', 'e' ]`
- First element: `x[0]`
- Nth element: `x[2]`
- Last element: `x[-1]`
- Next-to-last: `x[-2]`

# List Indexing

- Elements in a list are accessed by an index number.
- Index #'s start at 0.
- List: `x = [ 'a' , 'b' , 'c' , 'd' , 'e' ]`
- First element: `x[0] → 'a'`
- Nth element: `x[2] → 'c'`
- Last element: `x[-1] → 'e'`
- Next-to-last: `x[-2] → 'd'`

# List Slicing

- List: `x = ['a', 'b', 'c', 'd', 'e']`
- Slice syntax: `x[start:end:step]`
  - The start value is inclusive, the end value is exclusive.
  - Step is optional and defaults to 1.
  - Leaving out the end value means “go to the end”
  - Slicing always returns a **new list copied from the existing list**
- `x[0:1] → ['a']`
- `x[0:2] → ['a', 'b']`
- `x[-3:] → ['c', 'd', 'e']` # Third from the end to the end
- `x[2:5:2] → ['c', 'e']`



# List assignments and deletions

- Lists can have their elements overwritten or deleted (with the *del*) command.
- List: `x=['a', 'b', 'c', 'd', 'e']`
- `x[0] = -3.14` → `x` is now `[-3.14, 'b', 'c', 'd', 'e']`
- `del x[-1]` → `x` is now `[-3.14, 'b', 'c', 'd']`

# DIY Lists

- Go to the menu File→New File
- Enter your list commands there
- Give the file a name when you save it
- Use print() to print out results

- In the Spyder editor try the following things:
  - Assign some lists to some variables.
    - Try an empty list, repeated elements, initial set of elements
  - Add two lists: `a + b` What happens?
- Try list indexing, deletion, functions from *dir(my\_list)*
- Try assigning the result of a list slice to a new variable

# More on Lists and Variables

- Open the sample file *list\_variables.py* but don't run it yet!
- What do you think will be printed?
- Now run it...were you right?

```
x = ['a',[],'c',3.14]

y = x

# id() returns a unique identifier for a variable
print('x: %s      addr of x: %s' % (x,id(x)))
print('y: %s      addr of y: %s' % (y,id(y)))

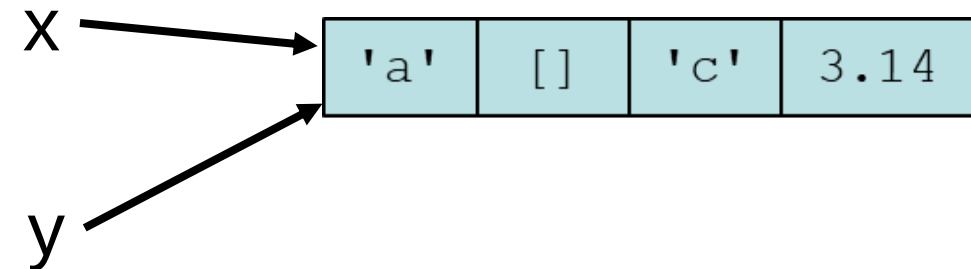
x[0] = -100

print('x: %s' % x)
print('y: %s' % y)
```

# Variables and Memory Locations

- Variables refer to a value stored in memory.
- `y = x` does **not** mean “make a copy of the list `x` and assign it to `y`” it means “make a copy of the memory location in `x` and assign it to `y`”
- `x` is **not the list** it's just a reference to it.

```
x = ['a', [], 'c', 3.14]  
y = x
```



# Copying Lists

```
z=x[:]  
z[0] = 'frog'  
print('x: %s      addr of x: %s' % (x,id(x)))  
print('z: %s      addr of z: %s' % (z,id(z)))
```

- How to copy (2 ways...there are more!):
- `y = x[:]` or `y=list(x)`
- In *list\_variables.py* uncomment the code at the bottom and run it.
- This behavior seems weird at first. It will make more sense when calling functions.

# Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- If / Else
- Lists
- Loops
- Functions

# While Loops

- While loops have a condition and a code block.
  - the indentation indicates what's in the while loop.
  - The loop runs until the condition is false.
- The *break* keyword will stop a while loop running.
- In the Spyder edit enter in some loops like these. Save and run them one at a time. What happens with the 1<sup>st</sup> loop?

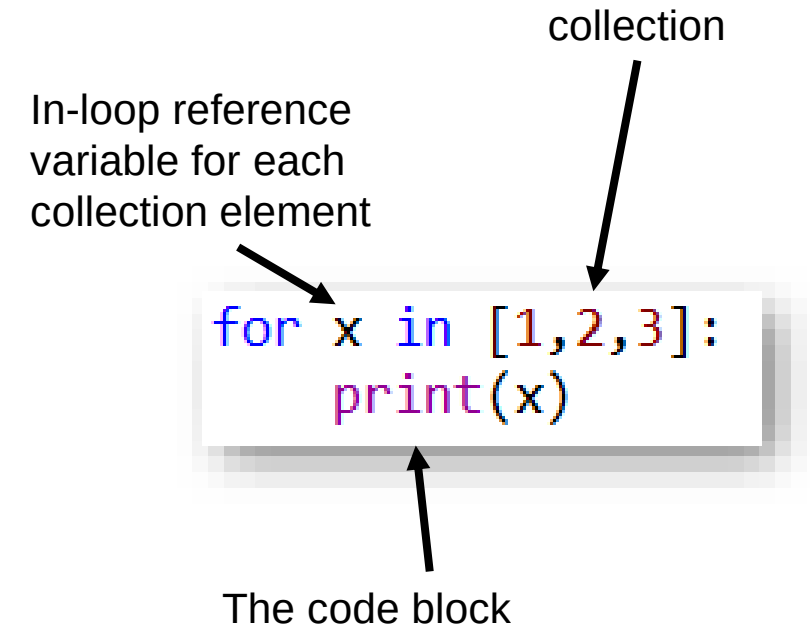
```
while True:
    print("looping!")

a=10
while a > 0:
    print(a)
    a -= 1

my_list=['a','b','c','d','e']
i=0
while i < len(my_list):
    print( my_list[i] )
    i += 1
    if i==3:
        break
```

# For loops

- *for* loops are a little different. They loop through a collection of things.
- The *for* loop syntax has a collection and a code block.
  - Each element in the collection is accessed in order by a reference variable
  - Each element can be used in the code block.
- The *break* keyword can be used in *for* loops too.





# Processing lists element-by-element

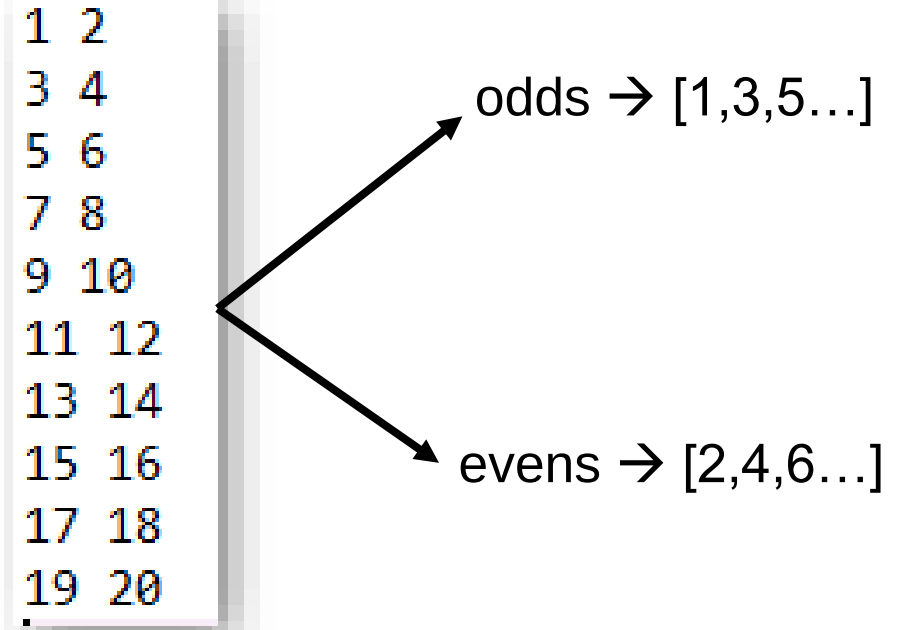
- A for loop is a convenient way to process every element in a list.
- There are several ways:
  - Loop over the list elements
  - Loop over a list of index values and access the list by index
  - Do both at the same time
  - Use a shorthand syntax called a *list comprehension*
- Open the file *looping\_lists.py*
- Let's look at code samples for each of these.

# The range() function

- The range() function auto-generates sequences of numbers that can be used for indexing into lists.
- Syntax: `range(start, exclusive end, increment)`
- `range(0,4)` → produces the sequence of numbers 0,1,2,3
- `range(-3,15,3)` → -3,0,3,6,9,12
- `range(4,-3,2)` → 4,2,0,-2
- Try this: `print(range(4))`

# Lists With Loops

- Open the file *read\_a\_file.py*
- This is an example of reading a file into a list. The file is shown to the right, *numbers.txt*
- We want to read the lines in the file into a list of strings (1 string for each line), then extract separate lists of the odd and even numbers.



- Edit *read\_a\_file.py* and try to figure this out.
- A solution is available in *read\_a\_file\_solved.py*
- Use the editor and run the code frequently after small changes!