

Introduction to Python

Part 1

v0.6.1

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
 - ~19,000 CPUs running Linux
- Consulting Focus:
 - Bioinformatics
 - Data Analysis / Statistics
 - Molecular modeling
 - Geographic Information Systems
 - Scientific / Engineering Simulation
 - Visualization
- CONTACT US: help@scc.bu.edu

SCC Python Tutorials

- Introduction to Python, Part one
- Introduction to Python, Part two
- Numerical and Scientific Computing in Python
- Python for Data Analysis
- Data Visualization in Python
- Introduction to Python Scikit-learn

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 onto **scc4** and use Python there.
 - Run:

```
module load python3/3.6.9  
/net/scc4/scratch/Intro_Python_code_0.4.sh  
cd ~/Intro_Python_code  
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.
- Be patient...it takes a while to start.



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/>
- [Intel Python](#)

BU's most popular personal 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 “end-of-life” mode – no new releases or patches after Dec 31, 2019
 - No library support (e.g. pandas, numpy, etc.) after that date
 - For more: <https://www.python.org/doc/sunset-python-2/>
- Py3 is not completely compatible with Py2
 - For learning Python these differences are almost negligible
- Which one to learn?
 - **Python 3**
 - 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

Latest SCC Python Versions

Python2

- python2/2.7.16
- python2-intel/2019.5.098

older

- python2/2.7.15
- python2-intel/2019.4.088

Python3

- python3/3.6.9 (default)
- python3/3.7.5
- python3-intel/2019.5.098

recommended
↙

older

- python3/3.6.5
- python3/3.7.3
- python3-intel/2019.4.088
- python3-intel/2020.0.014

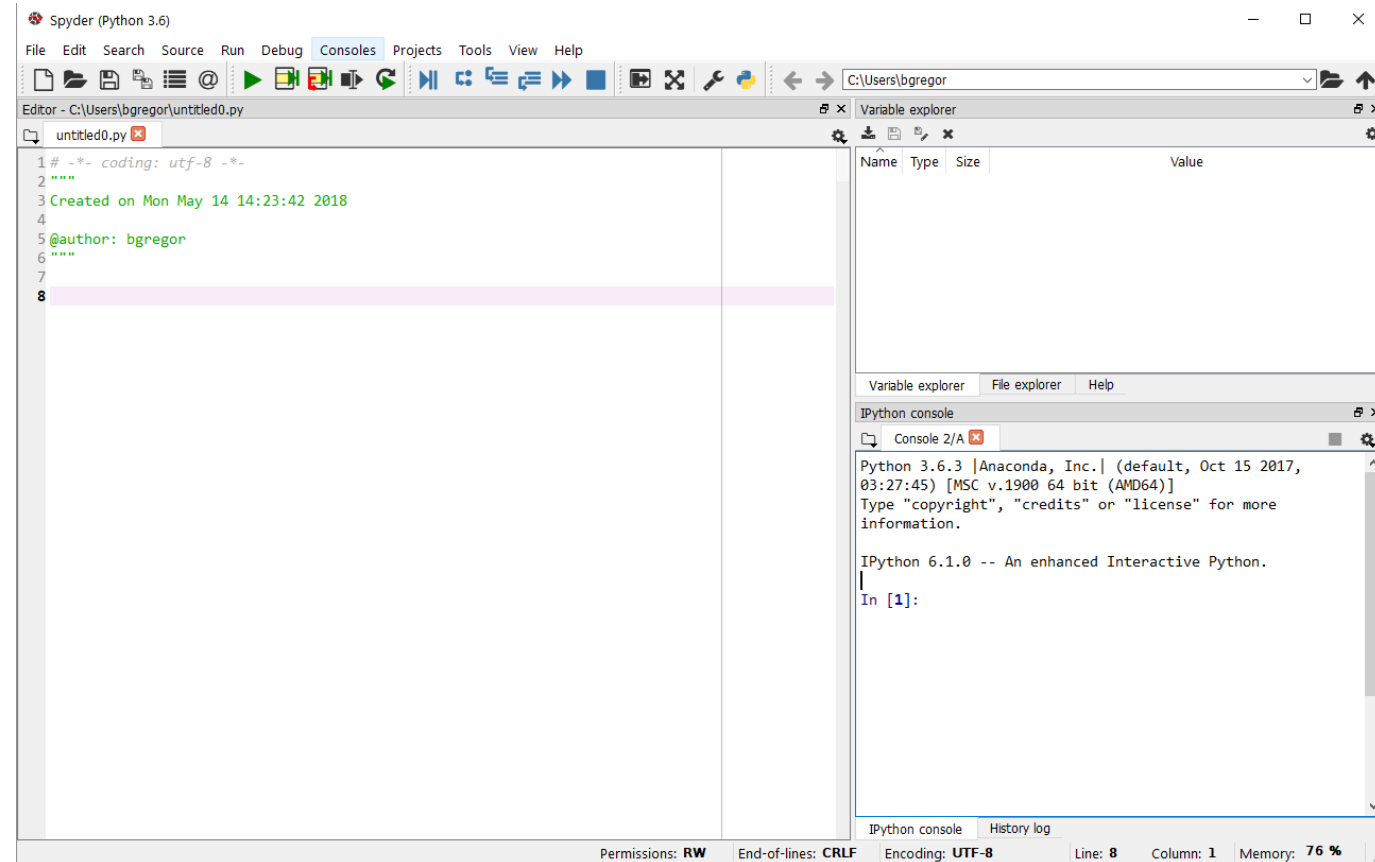
Spyder – a Python development environment

■ Pros:

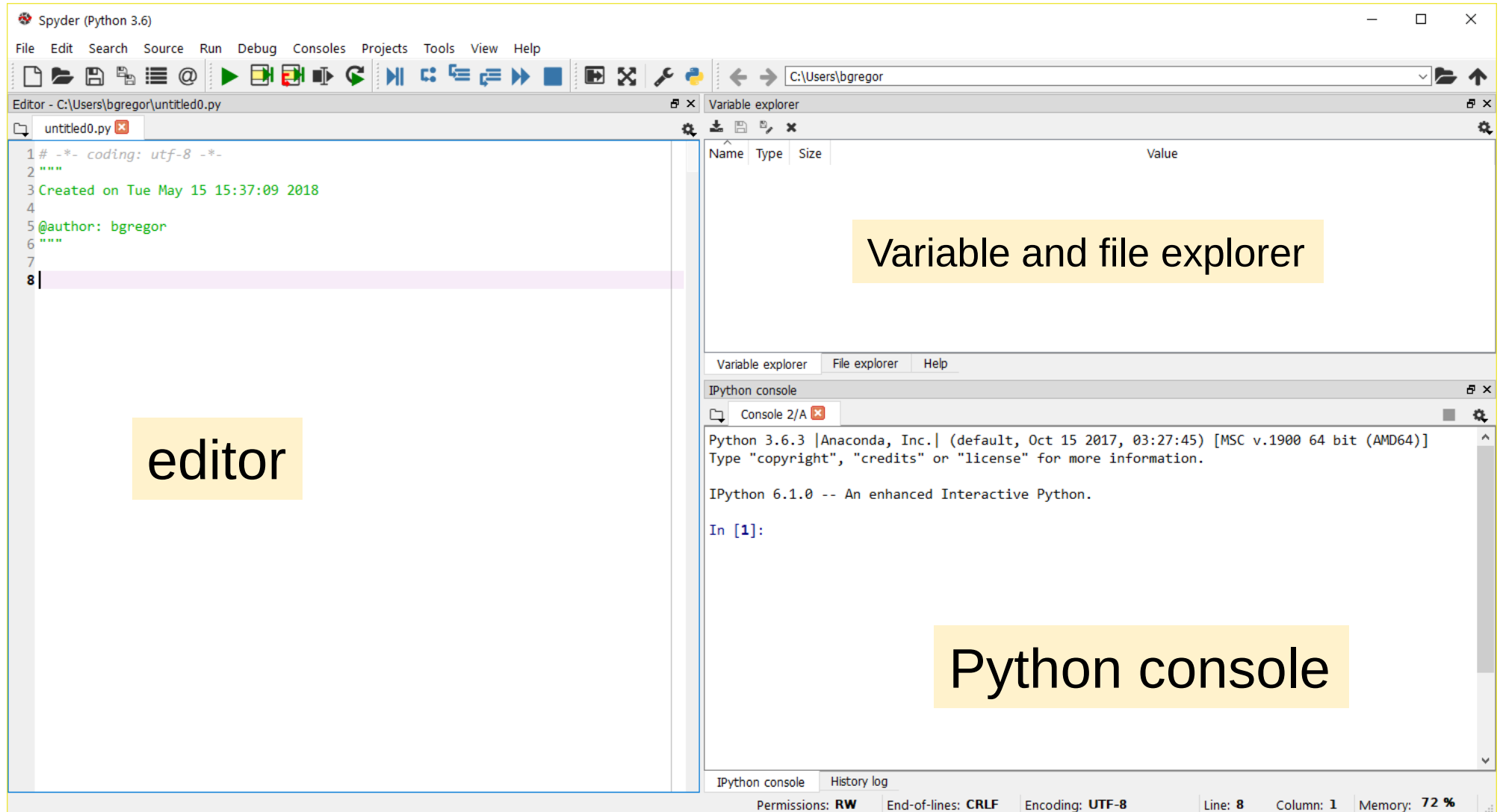
- Faster development
- Easier debugging!
- Helps organize code
- Increased efficiency

■ Cons

- Learning curve
- Can add complexity to smaller problems



The Spyder IDE



Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Classes
- If / Else

Tutorial Outline – Part 2

- Lists
- Tuples and dictionaries
- Modules
- numpy and matplotlib modules
- Script setup
- Development notes

Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Classes
- If / Else

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

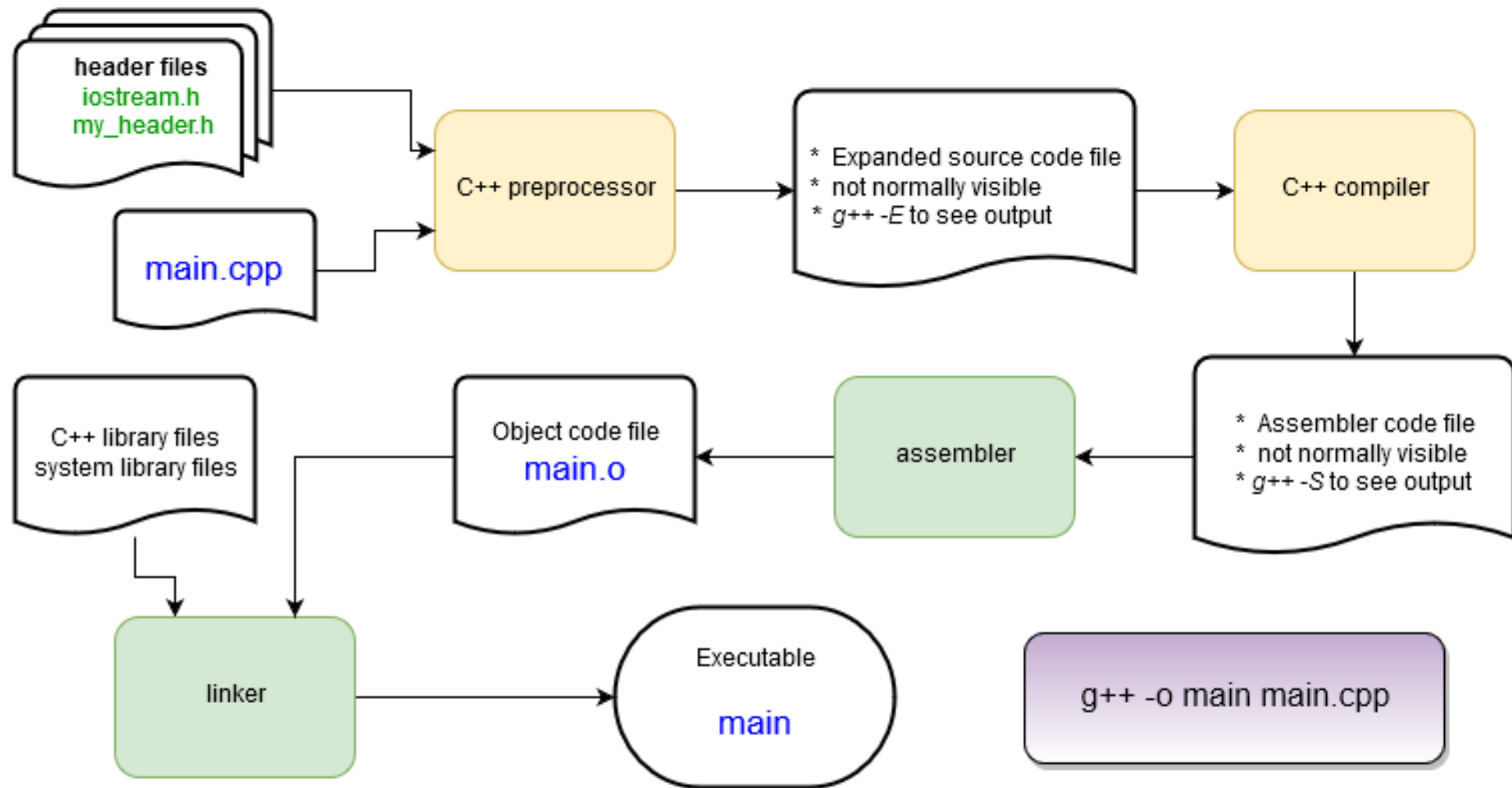
- “Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas...I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).”

—Python creator Guido Van Rossum, from the foreward to *Programming Python* (1st ed.)

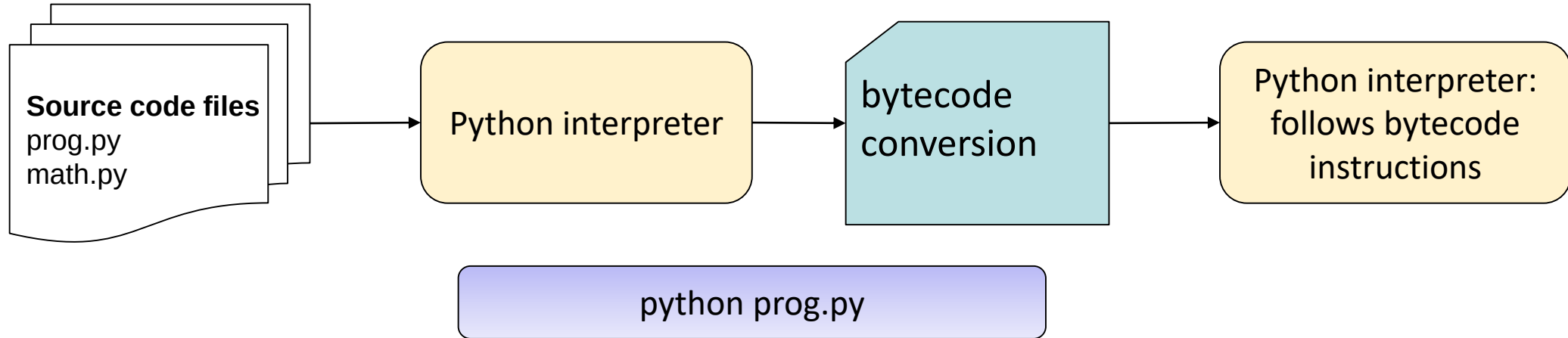
- Goals:
 - An easy and intuitive language just as powerful as major competitors
 - Open source, so anyone can contribute to its development
 - Code that is as understandable as plain English
 - Suitability for everyday tasks, allowing for short development times



Compiled Languages (ex. C++ or Fortran)



Interpreted Languages (ex. Python or R)



- A lot less work is done to get a program to start running compared with compiled languages!
- Python programs start running immediately – no waiting for the compiler to finish.
- 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
 - Compiler can optimize more extensively
- 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.

Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Classes
- If / Else

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
```

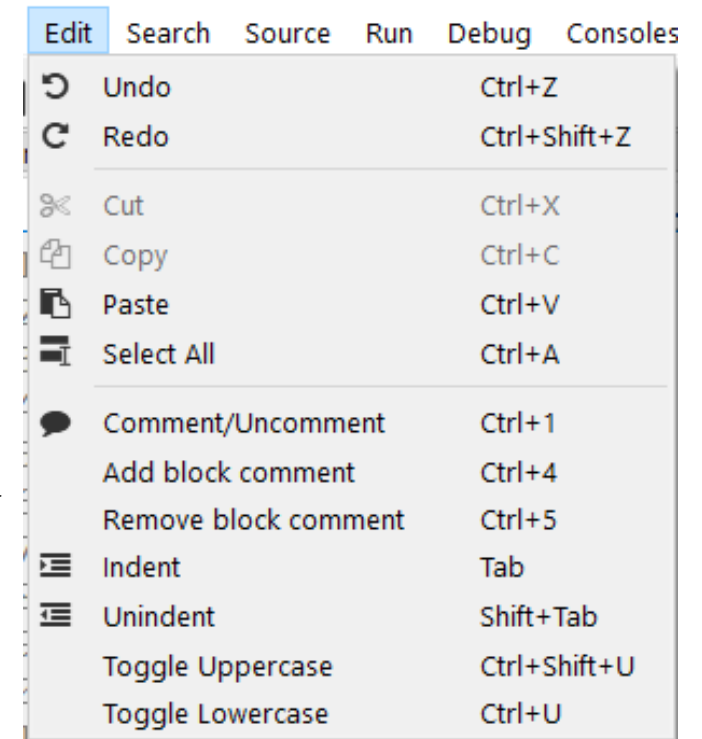
```
In [18]: True and (False or not False)  
Out[18]: True
```

```
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
- Functions
- Classes
- If / Else

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!
 - [Visualize Assignment](#)
- 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 - Z) 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, sets, and tuples**
 - These hold collections of variables
 - Specialty types: files, network connections, objects
- Custom types can be defined using Python classes.

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.

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

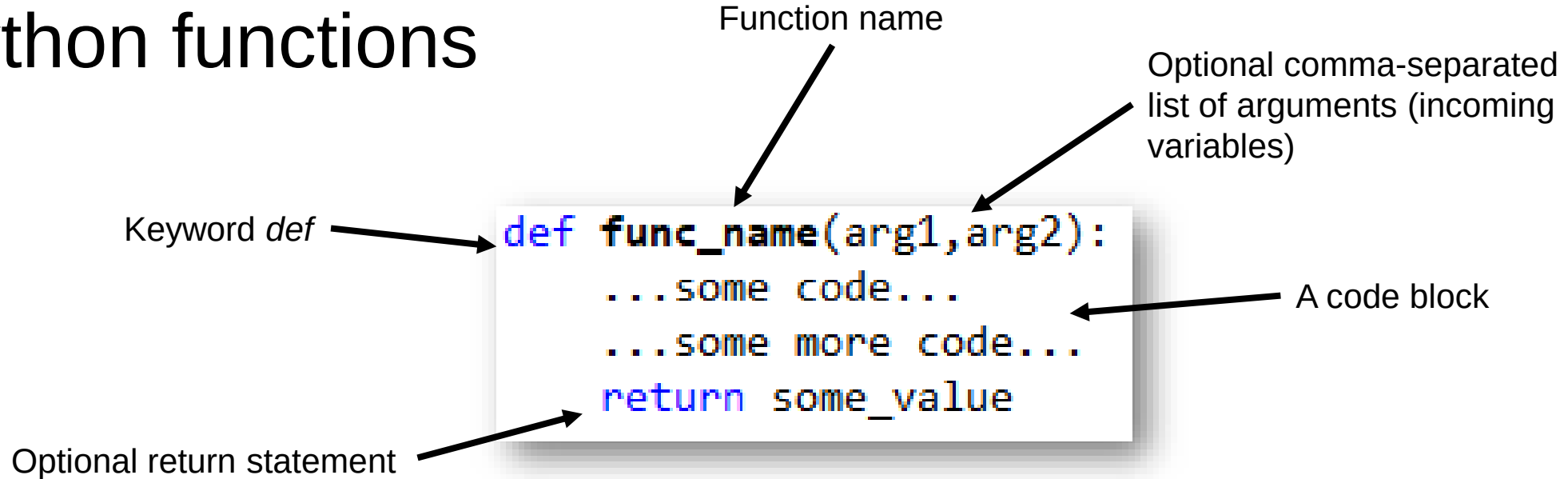
Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Classes
- If / Else

Functions

- Functions are used to create pieces of code that can be used in a program or in many programs.
- The use of functions is to logically separate a program into discrete computational steps.
- Programs that make heavy use of function definitions tend to be easier to:
 - develop
 - debug
 - maintain
 - understand

Python functions



- The return value can be any Python type
- If the return statement is omitted a special *None* value is still returned.
- The arguments are optional but the parentheses are required!
- **Functions must be defined** before they can be called.

Which code sample is easier to read?

■ C:

```
float avg(int a, int b, int c){  
    float sum = a + b + c ;  
    return sum / 3.0 ;}
```

or

```
float avg(int a, int b, int c)  
{  
    float sum = a + b + c ;  
    return sum / 3.0 ;  
}
```

■ Matlab:

```
function a = avg(x,y,z)  
    a = x + y + z ;  
    a = a / 3.0 ;  
end
```

or

```
function a = avg(x,y,z)  
    a = x + y + z ;  
    a = a / 3.0 ;  
end
```

Which code sample is easier to read?

- Most languages use special characters (`{ }` pairs) or keywords (*end*, *endif*) to indicate sections of code that belong to:
 - Functions
 - Control statements like *if*
 - Loops like *for* or *while*
- Python instead uses the **indentation** that programmers use anyway for readability.

C

```
float avg(int a, int b, int c)
{
    float sum = a + b + c ;
    return sum / 3.0 ;
}
```

Matlab

```
function a = avg(x,y,z)
    a = x + y + z ;
    a = a / 3.0 ;
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 :

```
def avg(x,y,z):  
    all_sum = x + y + z  
    return all_sum / 3.0
```

A code block

```
def mean(x,y,z):  
    return (x + y + z) / 3.0
```

Another code block

- This pattern is consistently repeated throughout Python syntax.
- Spaces or tabs can be mixed in a file but **not** within a code block.

Function Return Values

- A function can return any Python value.
- Function call syntax:

```
A = some_func()    # some_func returns a value
Another_func()     # ignore return value or nothing returned
b,c = multiple_vals(x,y,z)    # return multiple values
```

- Open *function_calls.py* for some examples

Function arguments

- Function arguments can be required or optional.
- Optional arguments are given a default value

```
def my_func(a, b, c=10, d=-1):  
    ...some code...
```

- To call a function with optional arguments:
- Optional arguments can be used in the order they're declared or out of order if their name is used.

```
my_func(x, y, z)           # a=x, b=y, c=z, d=-1  
my_func(x, y)              # a=x, b=y, c=10, d=-1  
my_func(x, y, d=w, c=z)    # a=x, b=y, c=z, d=w
```

Garbage collection

- Variables defined in a function no longer have any “live” references to them once the function returns.
- These variables become *garbage* – memory that can no longer be used.
- Python’s *garbage collection* operates to return that memory to the operating system so it can be re-used.
- There is no need to explicitly destroy or release most variables.
 - Some complex data types provide `.close()`, `.clean()`, etc. type functions. Use these where available.

When does garbage collection occur?

- It happens when Python thinks it should.
 - Part of the process is based on heuristics, i.e. tuned parameters in Python.
 - For the **great** majority of programs this is not an issue.
- Programs that use plenty of small functions tend to create and reclaim intermediate variables or results and memory usage is better controlled.
- The takeaway: use functions, and plenty of them.
- If you are having trouble with memory usage contact RCS for help!

Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Classes
- If / Else

Classes

- OOP: Object Oriented Programming
- In OOP a *class* is a data structure that combines data with functions that operate on that data.
- An *object* is a variable whose type is a *class*
 - Also called an *instance* of a class
- Classes provide a lot of power to help organize a program and can improve your ability to re-use your own code.

Object-oriented programming

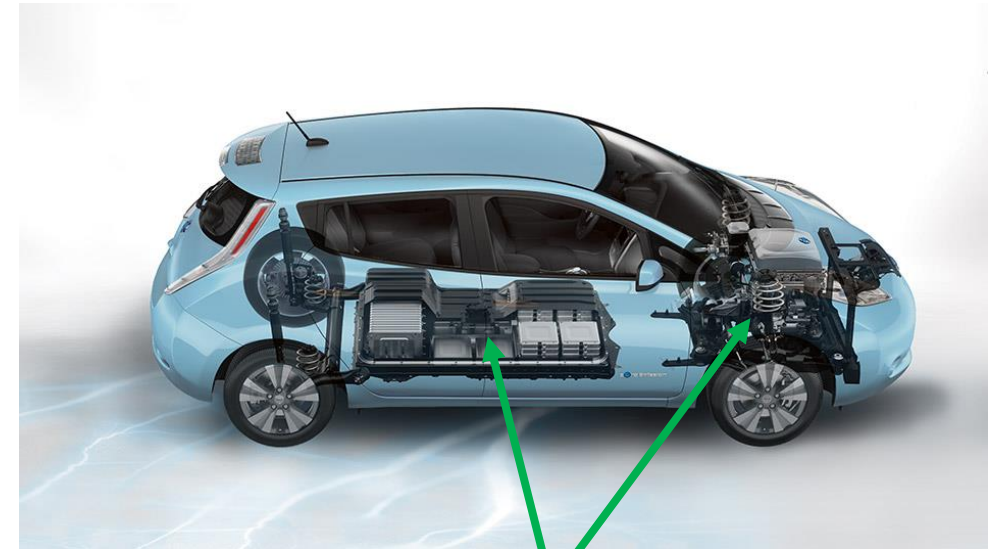
- Python is a fully object oriented programming (OOP) language.
- Object-oriented programming (OOP) seeks to define a program in terms of the *things* in the problem (files, molecules, buildings, cars, people, etc.), what they need, and what they can do.
- Some familiarity with OOP is needed to understand Python data structures and libraries.
- You can write your own Python classes to define custom data types.

Object-oriented programming

- OOP defines *classes* to represent the parts of the program.
- Classes can contain data and methods (internal functions).
- Classes can *inherit* from one another
 - A class (the subclass) can use all of the data and methods from another class (the superclass) and add its own.
- This is a highly effective way of modeling real world problems inside of a computer program.

“Class Car”

public interface



private data and methods

Syntax for using Python classes

Create an object, which is a variable whose type is a Python class.

Created by a call to the class or returned from a function.

Call a method for this object:

`object_name.method_name(args...)`

```
# Open a file. This returns a file object.
file = open('some_file.txt')

# Read a line from the text file.
line = file.readline()

# Get the filename
file.name # --> some_file.txt
```

Access internal data for this object:

`object_name.data_name`

Classes bundle data and functions

- In Python, calculate the area of some shapes after defining some functions.

```
radius = 14.0
width_square = 14.0
a1 = AreaOfCircle(radius)           # ok
a2 = AreaOfSquare(width_square)     # ok
a3 = AreaOfCircle(width_square)     # !! OOPS
```

- If we defined Circle and Rectangle classes with their own area() methods...it is not possible to miscalculate.

```
# create a Circle object with the radius value
c1 = Circle(radius)
r1 = Square(width_square)

a1 = c1.area()
a2 = r1.area()
```

Strings in Python

- Python defines a string class – **all strings in Python are objects.**
- This means strings have:
 - Their own internal (hidden) memory management to handle storage of the characters.
 - A variety of methods (functions) accessible once you have a string object in memory.
- You can't access string functions without a string – in Python the string provides its own functions.
 - No “strcat” / “strcmp” / etc as in C
 - No “strlen” / “isletter” / etc as in Matlab
 - No “nchar” / “toupper” /etc as in R

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)

```
mystr = 'Hello!'

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 object that contains **any** countable thing.

```
len(mystr) → 6
```

- 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.
 - These are defined in the string class as functions that operate on 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
```

%s means sub in value

variable name comes after a %

Variables are listed in the substitution order inside ()

- For more than one:

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

- Printing:

```
print('x: %s, y: %s, z: %s' % (xval, yval, 2.0))
```

String Substitutions (variations)

- There are 3 other ways to format strings...
- The “%s” approach is the original method.

- Alternative 1: [f-strings](#)

```
name = 'Boston'  
f'{name} University'
```

- Alternative 2: [string formatting](#)

```
"{} University".format("Boston")
```

- Alternative 3: String templates. Part of the string library.

```
from string import Template  
s = Template('$city $univ')  
s.substitute(city='Boston',  
             univ='University')
```

Writing Your Own Classes

```
class Student:
    def __init__(self, name, buid, gpa):
        self.name = name
        self.buid = buid
        self.gpa = gpa

    def has_4_0(self):
        return self.gpa==4.0

me = Student("RCS Instructor","U00000000",2.9)
print(me.has_4_0())
```

- Your own classes can be as simple or as complex as you need.
- Define your own Python classes to:
 - Bundle together logically related pieces of data
 - Write functions that work on specific types of data
 - Improve code re-use
 - Organize your code to more closely resemble the problem it is solving.

When to use your own class

- A class works best when you've done some planning and design work before starting your program.
- This is a topic that is best tackled after you're comfortable with solving programming problems with Python.
- Some tutorials on using Python classes:

W3Schools: https://www.w3schools.com/python/python_classes.asp

Python tutorial: <https://docs.python.org/3.6/tutorial/classes.html>

Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Classes
- If / Else

If / Else

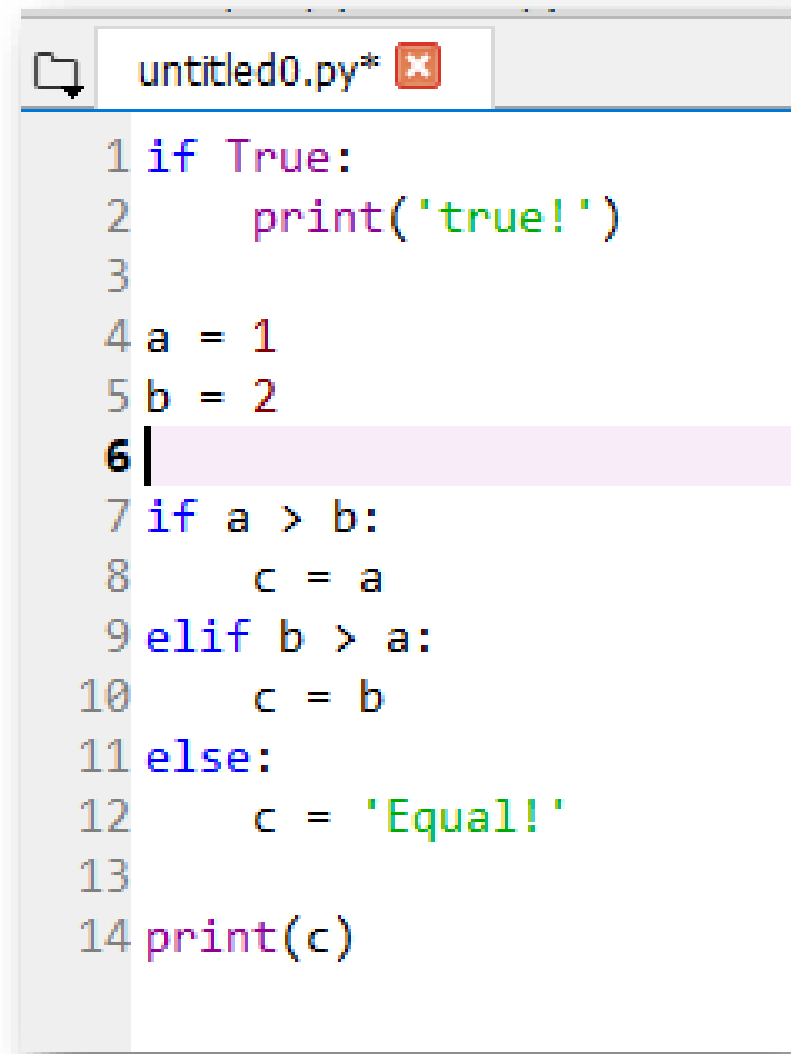
- *If*, *elif*, and *else* statements are used to implement conditional program behavior

- Syntax:

```
if Boolean_value:  
    ...some code  
elif Boolean_value:  
    ...some other code  
else:  
    ...more code
```

- *elif* and *else* are not required – use them 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?



The screenshot shows a Spyder Python IDE window titled 'untitled0.py*'. The code in the editor is as follows:

```
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)
```

Line 6 is highlighted with a light purple background, indicating a syntax error. The error is a missing colon at the end of the line.

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')
```



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.
- This sometimes causes confusion when pasting code into the console.
 - Solution: try to avoid pasting more than a few lines into the Python console.