Introduction to Python
Part 1

v0.3

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 anaconda3
spyder &
unzip /projectnb/scv/python/Intro_Python_code_0.3.zip
```
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/
  - https://python-xy.github.io/
BU’s most popular option: Anaconda

- [https://www.anaconda.com/download/](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 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
- Functions
- Classes
- If / Else
- Lists
Tutorial Outline – Part 2

- Loops
- Tuples and dictionaries
- Modules
- numpy and matplotlib modules
- Script setup
- Debugging
Tutorial Outline – Part 1

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

- **header files**: `iostream.h` `my_header.h`
- **C++ preprocessor**
  - Expanded source code file
  - Not normally visible
  - `g++ -E` to see output
- **C++ compiler**
- **Object code file**: `main.o`
- **Assembler**
  - Assembler code file
  - Not normally visible
  - `g++ -S` to see output
- **Linker**
- **Executable**
  - `main`

- `g++ -o main main.cpp`
Interpreted Languages (ex. Python or R)

- Source code files
  prog.py
  math.py

- Python interpreter

- Bytecode conversion

- Python interpreter: follows bytecode instructions

- 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

- Editor
- Python console
- Variable and file explorer
Tutorial Outline – Part 1

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

- Python supports a wide variety of operators which act like functions, i.e. they do something and return a value:
  - Arithmetic: $+$ $-$ $*$ $/$ $\%$ $**$
  - Logical: $\text{and}$ $\text{or}$ $\text{not}$
  - Comparison: $>$ $<$ $\geq$ $\leq$ $!=$ $==$
  - Assignment: $=$
  - Bitwise: $\&$ $|$ $\sim$ $^\wedge$ $\gg$ $\ll$
  - Identity: $\text{is}$ $\text{is not}$
  - Membership: $\text{in}$ $\text{not in}$
Try Python as a calculator

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

  \[ + \quad - \quad * \quad / \quad \% \quad ** \quad == \quad ( \quad ) \]

- Can you identify what they all do?
Try Python as a calculator

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

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division (Note: 3 / 4 is 0.75!)</td>
</tr>
<tr>
<td>%</td>
<td>Remainder (aka <em>modulus</em>)</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>==</td>
<td>Equals</td>
</tr>
</tbody>
</table>
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.
Tutorial Outline – Part 1

- What is Python?
- Operators
  - Variables
- Functions
- Classes
- If / Else
- Lists
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.
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: $x = \text{complex}(3,1)$ 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Effect</th>
<th>Equivalent to…</th>
</tr>
</thead>
<tbody>
<tr>
<td>x += y</td>
<td>Add the value of y to x</td>
<td>x = x + y</td>
</tr>
<tr>
<td>x -= y</td>
<td>Subtract the value of y from x</td>
<td>x = x - y</td>
</tr>
<tr>
<td>x *= y</td>
<td>Multiply the value of x by y</td>
<td>x = x * y</td>
</tr>
<tr>
<td>x /= y</td>
<td>Divide the value of x by y</td>
<td>x = x / y</td>
</tr>
</tbody>
</table>

- 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.
Tutorial Outline – Part 1

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

- Functions are used to create code that can be used in a program or in other programs.

- The use of functions to logically separate the program into discrete computational steps.

- Programs that make heavy use of function definitions tend to be easier to develop, debug, maintain, and 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.

```python
def func_name(arg1, arg2):
    ...some code...
    ...some more code...
    return some_value
```
Function Return Values

- A function can return any Python value.

- Function call syntax:

```python
A = some_func()      # return 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

```python
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.

```python
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
```
Function arguments

- Remember the list assignment?
  
  ```python
  x = ['a', [], 'c', 3.14]
y = x  # y points to the same list as x
  ```

- This applies in function calls too.
  
  ```python
  def my_func(a_list):
      # modifies the list in the calling routine!
      a_list.append(1)
  ```

- Then call it:
  
  ```python
  my_func(x)  # x and a_list inside the function are the same list!
  ```
Garbage collection

- Variables defined in a function (or in any code block) no longer have any “live” references to them once the function returns.

- These variables become *garbage*, and *garbage collection* operates to remove them from the computer’s memory, freeing up the memory to 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.
  - Simple data types (int, string, lists) will be taken care of automatically.
When does garbage collection occur?

- It happens when Python thinks it should.

- For the great majority of programs this is not an issue.

- Programs using very large quantities of memory or allocating large chunks of memory in repeated function calls can run into trouble.
Tutorial Outline – Part 1

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

- 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.

```
class GasMolecule:
    # Data:
    # - molecular weight, structure, common names, etc.
    # Methods:
    # - IR(wavenumStart, wavenumEnd) : return IR emission spectrum in range

GasMolecule ch4
GasMolecule co2
spectrum = ch4.IR(1000,3500)
Name = co2.common_name
```
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.
Encapsulation bundles data and functions

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

```python
# assume radius and width_square are assigned
# already
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.

```python
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 functions accessible once you have a string object in memory.

- You can’t access string functions without a string!
  - No “strcat” / “strcmp” / … as in C
  - No “strlen” / “isletter” / … as in Matlab
String functions

- In the Python console, create a string variable called `mystr`

- `type: dir(mystr)`

- Try out some functions:
  - `len(mystr)`
  - `mystr.upper()`
  - `mystr.title()`
  - `mystr.isdecimal()`

- Need help? Try:
  - `help(mystr.title)`
  - `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.

- 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.
String operators

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

```python
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)`

- %s means sub in value
- variable name comes after a %
- Variables are listed in the substitution order inside ()
Tutorial Outline – Part 1

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

- *If*, *elif*, and *else* statements are used to implement conditional program behavior.
- Syntax:
  ```python
  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?
Indentation of code...easier on the eyes!

- **C:**

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

- **Matlab:**

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

or

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

```matlab
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
```

- 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
- Functions
- Classes
- If / Else
- Lists
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!
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 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)`
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: \[ \text{x} = ['a', 'b', 'c', 'd', 'e'] \]

- First element: \( \text{x}[0] \rightarrow 'a' \)
- Nth element: \( \text{x}[2] \rightarrow 'c' \)
- Last element: \( \text{x}[-1] \rightarrow 'e' \)
- Next-to-last: \( \text{x}[-2] \rightarrow 'd' \)
List Slicing

- List: \( x = [\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \text{'e'}] \)
- 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

- \( x[0:1] \rightarrow [\text{'a'}] \)
- \( x[0:2] \rightarrow [\text{'a'}, \text{'b'}] \)
- \( x[-3:] \rightarrow [\text{'c'}, \text{'d'}, \text{'e'}] \)  # Third from the end to the end
- \( x[2:5:2] \rightarrow [\text{'c'}, \text{'e'}] \)
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` $\Rightarrow$ `x` is now `[-3.14, 'b', 'c', 'd', 'e']`

- `del x[-1]` $\Rightarrow$ `x` is now `[-3.14, 'b', 'c', 'd']`
DIY Lists

- 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 \( \text{dir(my\_list)} \)
  - Try assigning the result of a list slice to a new variable

- 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
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?
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.
- This is how all objects in Python are handled.
Copying Lists

- How to copy (2 ways…there are more!):
  - \( y = x[:] \) or \( y = \text{list}(x) \)
  - In *list_variables.py* uncomment the code at the bottom and run it.
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 1st loop?
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!