Introduction to Python

Introduction

About Me

My name is Sam Mishra.

I am a Graduate Student at NYU Tandon School of Engineering working on my Masters in Cybersecurity.

Email: sam640@nyu.edu

Introduction

Workshop Structure

> A whirlwind tour of the fundamentals of Python.
- Variables, Data Types, Control Flow, Functions, Object Oriented Programming, A Few Built-In Libraries, A Few Popular Math & Data Science 3rd Party Libraries

> Build some increasingly complicated applications (time permitting).

Workshop Goals

> Impart an appreciation for automation and scripting; Pique your curiosity of what you could potentially build with Python.

> Familiarize you with the fundamental components of Python.

> Make you comfortable with building a small Python script.

> Give you the tools to know how to keep learning Python - Where the documentation is and how to read it.

Objectives of this Lesson

1. Get Python Running
2. The Command-Line, Directories & Files
3. What is Python
4. Variables
5. Input/Output
6. Data Types - Numbers: int, float, complex
7. Data Types - Text: str
8. Data Types - Sequences: lists, tuples, range
9. Control Flow & Compound Statements: if, for while
10. Simple First Programs

Getting Python Running

Where To Get Python

Mac OSX:

https://www.python.org/downloads/mac-osx/

Windows:

https://www.python.org/downloads/windows/

Linux:

Debian/Ubuntu

```
sudo apt-get install python
sudo apt-get install python3.6
```

Fedora 21

```
sudo yum install python
sudo yum install python3
```

Fedora 22

```
sudo dnf install python
sudo dnf install python3
```

Setting Up Python - Adding Python to PATH

Windows:

> In Search, search for and then select: System (Control Panel)
> Click the **Advanced system settings** link.
> Click **Environment Variables**. In the section **System Variables**, find the PATH environment variable and select it. Click **New**.
> In the **New System Variable** window, specify the path to the Python directory (i.e. C:\Python).
> Click **OK**.

Linux / Mac OSX

> Depending on which shell you use you'll have to edit either ~/.profile, ~/.zshrc, or ~/.bash_profile using a command line text editor such as vim
> Add this line:

```
export PATH=$PATH:/path/to/python
```

> Save your changes

Getting Python Running

Running Python2 in MacOSX Terminal

```
slr:~ sammishra$ python
Python 2.7.13 (default, Dec 18 2016, 05:35:35)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello, world!'
>>> hello, world!
quit()
>>> slr:~ sammishra$
```

Running Python3 in MacOSX Terminal

```
slr:~ sammishra$ python3
Python 3.4.6 (default, Jan 20 2017, 08:18:10)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello, world!')
hello, world!
>>> quit()
slr:~ sammishra$
```

Python 2 vs Python 3:

> The main reason to use one over: some library you want to use is only supported by one of them.

> Python2 supports more libraries (many of which are useful and interesting) than Python3.

> Python3 is an improvement of Python2.

> It’s good to be comfortable with both.

> The differences aren’t huge and being familiar with one makes the it easy enough to figure out the other.

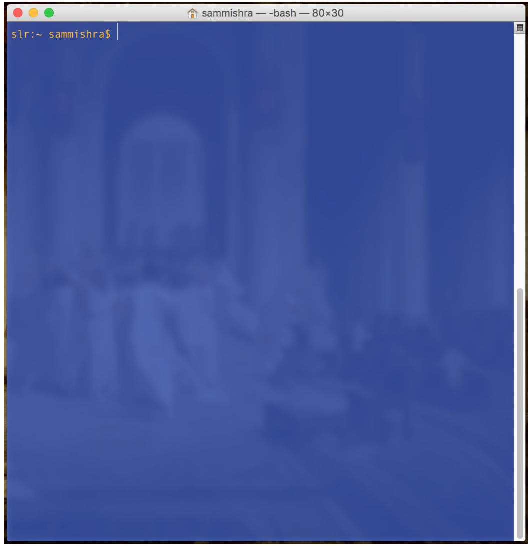
Read more here: https://wiki.python.org/moin/Python2orPython3

The Command-Line, Paths, Directories & Files

- A **Command-Line Interface**, a.k.a. a shell, is a user-interface that gives access to the operating system's services by typing in commands.

  On Windows: Command Prompt, PowerShell
  On Linux/MacOSX: Terminal, BASH

- This is opposed to a **Graphical User Interface**, which is the more common way of interfacing with an operating system, and uses things like a mouse, icons, windows, etc. to give access to the system's services.



The Command-Line, Paths, Directories & Files

- When dealing with the Command-Line, folders are referred to as **directories**.

- Directories have a hierarchical structure Linux/MacOSX directory structure begins from /. Windows normally starts from C:\. The full description of where a directory or file is is known as the **path**.
  For example: /Users/sammishra/Desktop/img.jpg

- In these directories are more directories or files. You can see the contents of a directory with the command -
  Linux/MacOSX: *ls*
  Windows*: dir*

- The shell always is ‘in’ one of these directories and can only operate on items in it. This is called the ‘working directory’. You can find out what the full path is to your current working directory is with the command -
  Linux/MacOSX: *pwd*

- The directory it is in can be changed by issuing a command -
  Linux/MacOSX: *cd*
  Windows*: chdir*

```
slr:~ sammishra$ pwd
/Users/sammishra
slr:~ sammishra$ ls
Applications    Downloads       Music           VirtualBox VMs  macports
Desktop         Library         Pictures        default_prx     socket
Documents       Movies          Public          gnuradio
slr:~ sammishra$ cd Music/
slr:Music sammishra$ ls
iTunes
slr:Music sammishra$ cd iTunes/
slr:iTunes sammishra$ ls
Album Artwork                   iTunes Library.itl
Previous iTunes Libraries       iTunes Media
iTunes Library Extras.itdb      sentinel
iTunes Library Genius.itdb
slr:iTunes sammishra$ pwd
/Users/sammishra/Music/iTunes
slr:iTunes sammishra$ cd /Users/sammishra/
slr:~ sammishra$ cd /
slr:/ sammishra$ pwd
/
slr:/ sammishra$ ls
Applications                    home
Library                         installer.failurerequests
Network                         model
System                          net
Users                           opt
Volumes                         private
bin                             sbin
cores                           tmp
dev                             usr
etc                             var
slr:/ sammishra$ cd Users/
slr:Users sammishra$ ls
Shared          sammishra
slr:Users sammishra$ pwd
/Users
slr:Users sammishra$ cd ..
slr:/ sammishra$ pwd
/
slr:/ sammishra$
```

What is Python

According to Wikipedia:

Python is a **high-level programming language** for **general-purpose programming**.

An **interpreted language**, Python has a design philosophy which emphasizes **code readability**, and a syntax which allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java.

Python features a dynamic type system and automatic memory management.

* All this means you don’t have to deal with or understand the ins and outs of how a computer operates to get your code running

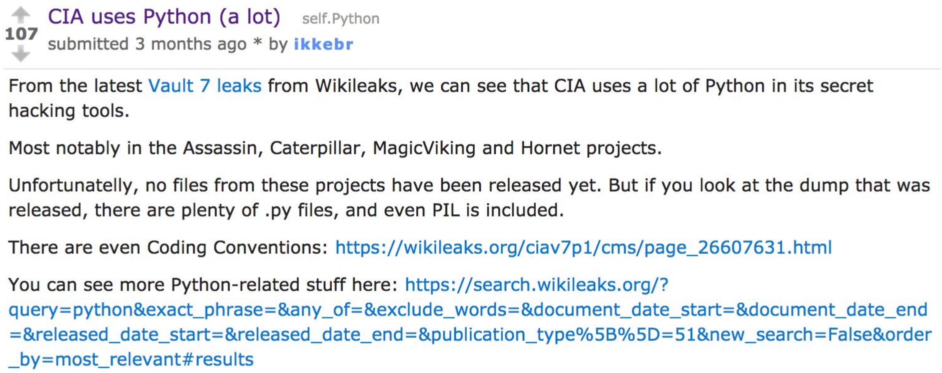
What is Python

Python is used by everyone. From the hacking tools of the CIA/NSA to the high frequency trading tools used by financial companies.

Automation and scripting are powerful tools to have at your disposal, especially for processing data.

Python is an especially good first computer programming language to learn because:

- scripts almost read like english
- it is useful and usually “enough” - the advantages of languages like C and Java aren’t always necessary and often not worth the headache
- it requires not much, if any, knowledge of computer science,
- it prepares you to learn the lower-level languages



107 CIA uses Python (a lot) self.Python

submitted 3 months ago * by ikkebr

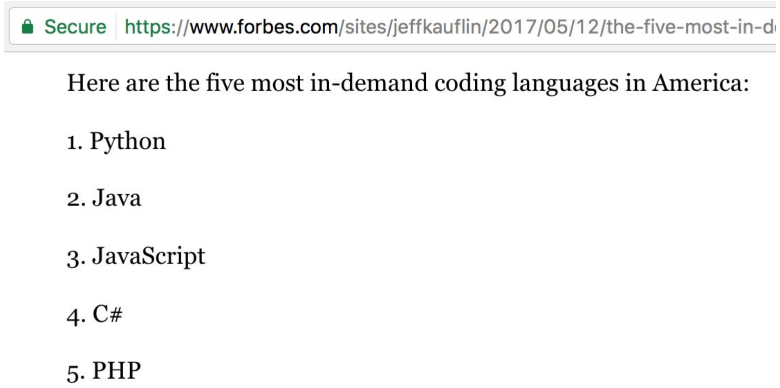
From the latest Vault 7 leaks from Wikileaks, we can see that CIA uses a lot of Python in its secret hacking tools.

Most notably in the Assassin, Caterpillar, MagicViking and Hornet projects.

Unfortunatelly, no files from these projects have been released yet. But if you look at the dump that was released, there are plenty of .py files, and even PIL is included.

There are even Coding Conventions: https://wikileaks.org/ciav7p1/cms/page_26607631.html

You can see more Python-related stuff here: https://search.wikileaks.org/?query=python&exact_phrase=&any_of=&exclude_words=&document_date_start=&document_date_end=&released_date_start=&released_date_end=&publication_type%5B%5D=51&new_search=False&order_by=most_relevant#results



Secure https://www.forbes.com/sites/jeffkauflin/2017/05/12/the-five-most-in-d

Here are the five most in-demand coding languages in America:

1. Python

2. Java

3. JavaScript

4. C#

5. PHP

What is Python

The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

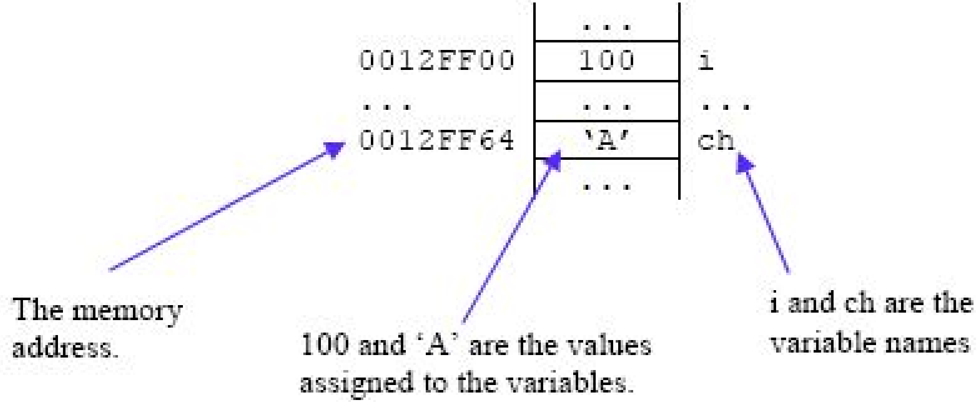
Objectives of this Lesson

1. ~~Get Python Running~~
2. ~~The Command-Line, Directories & Files~~
3. ~~What is Python~~
4. Variables
5. Input/Output
6. Data Types - Numbers: int, float, complex
7. Data Types - Text: str
8. Data Types - Sequences: lists, tuples, range
9. Control Flow & Compound Statements: if, for while
10. Simple First Programs

Variables

In computer programming, a **variable** is a **storage location** *paired with an associated* **symbolic name** (an **identifier**), which contains some known or unknown *quantity of information* referred to as a **value**.

The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents.



Variables

<u>Rules for Naming Variables in Python:</u>

[+] Starts with: a letter *A* to *Z*, or *a* to *z*, or an underscore _

[+] Followed by: zero or more letters, underscores and digits

[+] Punctuation characters [@,$,%] not allowed

[+] Case sensitive - x and X are different

<u>Conventions for Naming Variable in Python:</u>

[+] Readability is very important.

- python_puppet
- pythonpuppet
- pythonPuppet

[+] Descriptive names are very useful.

```
Python 3.4.6 (default, Jan 20 2017, 08:18:10)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 7
>>> y = 12
>>> john = 'a generic person'
>>> john_doe = 'a slightly less generic person'
>>> x
7
>>> y
12
>>>
>>> john
'a generic person'
>>> john_doe
'a slightly less generic person'
>>> jane_doe
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'jane_doe' is not defined
>>> 7IsLucky = 'i cant do this'
  File "<stdin>", line 1
    7IsLucky = 'i cant do this'
           ^
SyntaxError: invalid syntax
>>> 7 = x
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> and = '&'
  File "<stdin>", line 1
    and = '&'
      ^
SyntaxError: invalid syntax
>>> True = 'blue'
  File "<stdin>", line 1
SyntaxError: can't assign to keyword
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', '
def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'retu
rn', 'try', 'while', 'with', 'yield']
```

Input/Output

input([*prompt*])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

```
slr:~ sammishra$ python3
Python 3.4.6 (default, Jan 20 2017, 08:18:10)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = input('When you see this type something > ')
When you see this type something > which Witch is which
>>> print(x)
which Witch is which
```

standard streams are input and output communication channels between a computer program and its environment of execution.
The three I/O connections are called **standard input** (**stdin**), **standard output** (**stdout**) and **standard error** (**stderr**).
Originally I/O happened via a physically connected system (input via keyboard, output via monitor), but standard streams **abstract** this.
When a command is executed via a shell, the streams are typically connected to the text terminal on which the shell is running.

Objectives of this Lesson

1. ~~Get Python Running~~
2. ~~The Command-Line, Directories & Files~~
3. ~~What is Python~~
4. ~~Variables~~
5. ~~Input/Output~~
6. Data Types - Numbers: int, float, complex
7. Data Types - Text: str
8. Data Types - Sequences: lists, tuples, range
9. Control Flow & Compound Statements: if, for while
10. Simple First Programs

Data Types

Mutability vs. Immutability

- In general, data types in Python can be distinguished based on whether objects of the type are **mutable** or **immutable**.

- The content of objects of immutable types cannot be changed after they are created.

- Only mutable objects support methods that change the object in place, such as reassignment of a sequence slice, which will work for lists, but raise an error for tuples and strings.

class **type**(*object*)

With one argument, return the type of an *object*.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

```
>>> type(45)
<class 'int'>
>>> type('word')
<class 'str'>
>>> type(9.99)
<class 'float'>
>>> type([1,2,3])
<class 'list'>
>>> type((3,5))
<class 'tuple'>
>>> x = 7
>>> y = 'red'
>>> type(x)
<class 'int'>
>>> type(y)
<class 'str'>
```

Data Types

Boolean (True or False)

Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

```
>>> True == 1
True
>>> False == 0
True
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

```
>>> not True
False
>>> not False
True
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

Comparison Operators

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Comparison operators all have the same priority (which is higher than that of the Boolean operations).

Data Types - Numbers

There are three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*.

- **int (signed integers)**: They are often called just integers or ints, are positive or negative whole numbers with no decimal point.

- **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).

- **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. To extract these parts from a complex number z, use `z.real` and `z.imag`

All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

Data Types - Numbers

```
Python 3.4.6 (default, Jan 20 2017, 08:18:10)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 12 + 37
49
>>> 22.4 + 19.1
41.5
>>> type(12 + 37)
<class 'int'>
>>> type(22.4 + 19.1)
<class 'float'>
>>> 12 + 19.1
31.1
>>> type(12 + 19.1)
<class 'float'>
>>> 12 + 12.0
24.0
>>> type(12 + 12.0)
<class 'float'>
>>> type('12')
<class 'str'>
>>> int('12')
12
>>> type(int('12'))
<class 'int'>
>>> float('12')
12.0
>>> type(float('12'))
<class 'float'>
>>> type(99)
<class 'int'>
>>> type(99.0)
<class 'float'>
>>> float(99)
99.0
>>> int(97.9)
97
```

```
Python 3.4.6 (default, Jan 20 2017, 08:18:10)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 15
>>> y = 18
>>> z = 5.6
>>> type(x)
<class 'int'>
>>> type(y)
<class 'int'>
>>> type(z)
<class 'float'>
>>> x-y
-3
>>> x*y
270
>>> y/x
1.2
>>> type(y/x)
<class 'float'>
>>> y//x
1
>>> y%x
3
>>> divmod(y,x)
(1, 3)
>>> abs(x-y)
3
>>> x**2
225
>>> pow(x,2)
225
>>> x/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Data Types - Text

Textual data in Python is handled with str objects, or **strings**. Strings are **immutable** sequences of Unicode code points. String literals are written in a variety of ways:

- Single quotes: 'allows embedded "double" quotes'
- Double quotes: "allows embedded 'single' quotes".
- Triple quoted: '''Three single quotes''', """Three double quotes"""

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

var1 = 'Hello'

```
 H   e   l   l   o
 0   1   2   3   4
-5  -4  -3  -2  -1
```

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

Data Types - Text

```
Python 3.4.6 (default, Jan 20 2017, 08:18:10)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> string = '''All work and no play makes Jack a dull boy
... All wwork and no play makes Jack a dull boy
... All work and no play makes jack adulll bot
... All work and no play makes Jack a dull boy
... All work and no play makes Jack a dull boy'''
>>> string
'All work and no play makes Jack a dull boy\nAll wwork and no play makes Jack a dull boy\nAll work and no play makes jack adulll bot\nAll work and no play makes Jack a dull boy\nAll work and no play makes Jack a dull boy'
>>> print(string)
All work and no play makes Jack a dull boy
All wwork and no play makes Jack a dull boy
All work and no play makes jack adulll bot
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
>>> string1 = 'Hello, '
>>> string2 = 'World!'
>>> string1 + string2
'Hello, World!'
>>> string1 * 3
'Hello, Hello, Hello, '
>>> string1 * 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

```
>>> len(string1)
7
>>> string1[0]
'H'
>>> string1[1:4]
'ell'
>>> string1[-3]
'o'
>>> string1[-2:]
', '
>>> string3 = string1 + string2
>>> string3
'Hello, World!'
```

Data Types - Text

Escape Sequence	Meaning
newline	Ignored
\\\\	Backslash (\\)
\\'	Single quote (')
\\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
ooo	ASCII character with octal value *ooo*
\x*hh...*	ASCII character with hex value *hh...*

String Formatting

str.**format**(**args*, ***kwargs*)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

String Formatting Operator: %

```
print "My name is %s and weight is %d kg!" % ('Zara', 21)
'My name is Zara and weight is 21 kg!'
```

Data Types - Text

```
>>> 'Hello! My Name is {first} {last}'.format(first='Sam', last='Mishra')
'Hello! My Name is Sam Mishra'
>>> 'Hello! My Name is {} {}'.format('Sam', 'Mishra')
'Hello! My Name is Sam Mishra'
>>> 'First: {} Last: {}'.format('Sam', 'Mishra')
'First: Sam Last: Mishra'
>>> 'First: {} \nLast: {}'.format('Sam', 'Mishra')
'First: Sam \nLast: Mishra'
>>> print('First: {} \nLast: {}'.format('Sam', 'Mishra'))
First: Sam
Last: Mishra
```

```
>>> print('First: \t{} \nLast: \t{}'.format('Sam', 'Mishra'))
First:  Sam
Last:   Mishra
>>> print("Simon says "jump" Simon says "sit" "bark" Simon didn't say")
  File "<stdin>", line 1
    print("Simon says "jump" Simon says "sit" "bark" Simon didn't say")
                          ^
SyntaxError: invalid syntax
>>> print("Simon says 'jump' Simon says 'sit' 'bark' Simon didn't say")
Simon says 'jump' Simon says 'sit' 'bark' Simon didn't say
>>> print("Simon says \"jump\" Simon says \"sit\" \"bark\" Simon didn't say")
Simon says "jump" Simon says "sit" "bark" Simon didn't say
```

```
>>> phone_number_template = '{} - {} - {}'
>>> phone_number_template.format(555,123,4567)
'555 - 123 - 4567'
>>> phone_number_template.format(555,123,4567,890)
'555 - 123 - 4567'
>>> phone_number_template.format(555,123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:*<30}'.format('left aligned')
'left aligned******************'
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:x>30}'.format('right aligned')
'xxxxxxxxxxxxxxxxxright aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:_^30}'.format('centered')
'___________centered___________'
```

Data Types - Sequence Types

There are three basic sequence types: **lists**, **tuples**, and **range** objects.

The operations in the following table are supported by most sequence types, both mutable and immutable. This table lists the sequence operations sorted in ascending priority.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+`(concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Data Types - Sequence Types

Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the hash() built-in.

Mutable Sequence Types

In the table *s* is an instance of a mutable sequence type, *t* is any iterable object and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

Data Types - Sequence: Lists

Lists are **mutable** sequences, typically used to store collections of homogeneous items.

class **list**([*iterable*])

Lists may be constructed in several ways:
Using a pair of square brackets to denote the empty list: []
Using square brackets, separating items with commas: [a], [a, b, c]
Using a list comprehension: [x for x in iterable]
Using the type constructor: list() or list(iterable)

Lists implement all of the common and mutable sequence operations. Lists also provide the following additional method:

list_obj =	[‘a’ ,	‘b’ ,	‘c’ ,	‘d’ ,	‘e’]
index:	0	1	2	3	4
Index:	-5	-4	-3	-2	-1

sort(*, *key=None*, *reverse=None*)

This method sorts the list in place, using only < comparisons between items.

Data Types - Sequence: Lists

```
>>> list1 = [1, 2, 3]
>>> list2 = [3, 1, 2]
>>> list1 == list2
False
```

```
>>> nums
[2, 5, 6, 6, 7]
>>> min(nums)
2
>>> max(nums)
7
>>> nums[0:len(nums)] = [0] * len(nums)
>>> nums
[0, 0, 0, 0, 0]
>>> nums[0:len(nums):2] = [1] * (len(nums)//2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 2 to extended slice of size 3
>>> nums[1:len(nums):2] = [1] * (len(nums)//2)
>>> nums
[0, 1, 0, 1, 0]
```

Data Types - Sequence: Lists

```
>>> dead_prez = ['Washington', 'Adams', 'Jefferson', 'Madison', 'Monroe', 'Adams
', 'Jackson', 'Van Buren', 'Harrison', 'Tyler', 'Polk', 'Taylor', 'Fillmore', 'P
ierce', 'Buchanan', 'Lincoln', 'Johnson', 'Grant', 'Hayes', 'Garfield', 'Arthur'
, 'Cleveland', 'Harrison', 'McKinley', 'Roosevelt', 'Taft', 'Wilson', 'Harding',
 'Coolidge', 'Hoover', 'Roosevelt', 'Truman', 'Eisenhower', 'Kennedy', 'Johnson'
, 'Nixon', 'Ford', 'Reagan']
>>> len(dead_prez)
38
>>> former_prez = ['Carter', 'Bush', 'Clinton', 'Bush', 'Obama']
>>> len(former_prez)
5
>>> potus = ['Trump']
>>> len(potus)
1
>>> us_prez = dead_prez + former_prez + potus
>>> len(us_prez)
44
>>> # Silly Grover Cleveland Is Not Important Enough To Count Twice
...
>>> us_prez
['Washington', 'Adams', 'Jefferson', 'Madison', 'Monroe', 'Adams', 'Jackson', 'V
an Buren', 'Harrison', 'Tyler', 'Polk', 'Taylor', 'Fillmore', 'Pierce', 'Buchana
n', 'Lincoln', 'Johnson', 'Grant', 'Hayes', 'Garfield', 'Arthur', 'Cleveland', '
Harrison', 'McKinley', 'Roosevelt', 'Taft', 'Wilson', 'Harding', 'Coolidge', 'Ho
over', 'Roosevelt', 'Truman', 'Eisenhower', 'Kennedy', 'Johnson', 'Nixon', 'Ford
', 'Reagan', 'Carter', 'Bush', 'Clinton', 'Bush', 'Obama', 'Trump']
>>> double_trouble = potus * 2
>>> double_trouble
['Trump', 'Trump']
>>> huh = potus * 'f'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

```
>>> us_prez[0]
'Washington'
>>> us_prez[44]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> us_prez[43]
'Trump'
>>> max(us_prez)
'Wilson'
>>> min(us_prez)
'Adams'
>>> us_prez.count('Bush')
2
>>> us_prez.count('Roosevelt')
2
>>> us_prez.index['Obama']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'builtin_function_or_method' object is not subscriptable
>>> us_prez.index('Obama')
42
>>> # Lets Add Cleveland's Second Term For Accuracy :(
...
>>> us_prez.insert(23, 'Cleveland')
>>> us_prez
['Washington', 'Adams', 'Jefferson', 'Madison', 'Monroe', 'Adams', 'Jackson', 'V
an Buren', 'Harrison', 'Tyler', 'Polk', 'Taylor', 'Fillmore', 'Pierce', 'Buchana
n', 'Lincoln', 'Johnson', 'Grant', 'Hayes', 'Garfield', 'Arthur', 'Cleveland', '
Harrison', 'Cleveland', 'McKinley', 'Roosevelt', 'Taft', 'Wilson', 'Harding', 'C
oolidge', 'Hoover', 'Roosevelt', 'Truman', 'Eisenhower', 'Kennedy', 'Johnson', '
Nixon', 'Ford', 'Reagan', 'Carter', 'Bush', 'Clinton', 'Bush', 'Obama', 'Trump']
>>> us_prez.index('Obama')
43
```

Data Types - Sequence: Lists

```
>>> queue = []
>>> queue.insert(0,'Alice')
>>> queue
['Alice']
>>> queue.insert(0,'Bob')
>>> queue
['Bob', 'Alice']
>>> queue.insert(0,'Nancy')
>>> queue.insert(0,'Larry')
>>> queue
['Larry', 'Nancy', 'Bob', 'Alice']
>>> next = queue.pop()
>>> next
'Alice'
>>> queue
['Larry', 'Nancy', 'Bob']
>>> next = queue.pop()
>>> next
'Bob'
>>> queue
['Larry', 'Nancy']
>>> queue.insert(0,'Roger')
>>> qu
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'qu' is not defined
>>> queue
['Roger', 'Larry', 'Nancy']
```

```
>>> line_cutter = 'Sally'
>>> queue.append(line_cutter)
>>> queue
['Roger', 'Larry', 'Nancy', 'Sally']
>>> next = queue.pop()
>>> next
'Sally'
>>> queue
['Roger', 'Larry', 'Nancy']
>>> queue.insert(0,line_cutter)
>>> queue
['Sally', 'Roger', 'Larry', 'Nancy']
>>> queue.remove('Sally')
>>> queue
['Roger', 'Larry', 'Nancy']
```

Data Types - Sequence: Tuples

Tuples are **immutable sequences**, typically used to store collections of heterogeneous data. Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a set or dict instance).

class **tuple**([*iterable*])

Tuples may be constructed in a number of ways:
Using a pair of parentheses to denote the empty tuple: `()`
Using a trailing comma for a singleton tuple: `a,` or `(a,)`
Separating items with commas: `a, b, c` or `(a, b, c)`
Using the tuple() built-in: `tuple()` or `tuple(iterable)`

Tuples implement all of the common sequence operations.

Data Types - Sequence: Tuples

```
>>> obama = ('Barack', 'Obama', 44, 'Dem', 'Illinois')
>>> obama[0:2]
('Barack', 'Obama')
>>> hash(obama)
-5993085273964571298
>>> clinton = ('Bill', 'Clinton', 42, 'Dem', 'Arkansas')
>>> hash(clinton)
-3407707242767842665
>>> obama == clinton
False
>>> hash(obama)
-5993085273964571298
>>> obama == obama
True
>>> dems = obama + clinton
>>> dems
('Barack', 'Obama', 44, 'Dem', 'Illinois', 'Bill', 'Clinton', 42, 'Dem', 'Arkans
as')
>>> dems[0:10:5]
('Barack', 'Bill')
>>>
```

Data Types - Sequence: Range

The range type represents an **immutable sequence of numbers** and is commonly used for looping a specific number of times in for loops.

class **range**(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers. If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, ValueError is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start +step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

Ranges implement all of the common sequence operations except concatenation and repetition.

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Data Types - Sequence: Range

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Objectives of this Lesson

1. Get Python Running
2. ~~The Command-Line, Directories & Files~~
3. ~~What is Python~~
4. ~~Variables~~
5. ~~Input/Output~~
6. ~~Data Types - Numbers: int, float, complex~~
7. ~~Data Types - Text: str~~
8. ~~Data Types - Sequences: lists, tuples, range~~
9. Control Flow & Compound Statements: if, for while
10. Simple First Programs

Control Flow

In computer science, **control flow** (or **flow of control**) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

Within an imperative programming language, a *control flow statement* is a statement which execution results in a choice being made as to which of two or more paths to follow.

The **if, while** and **for** statements implement traditional control flow constructs.

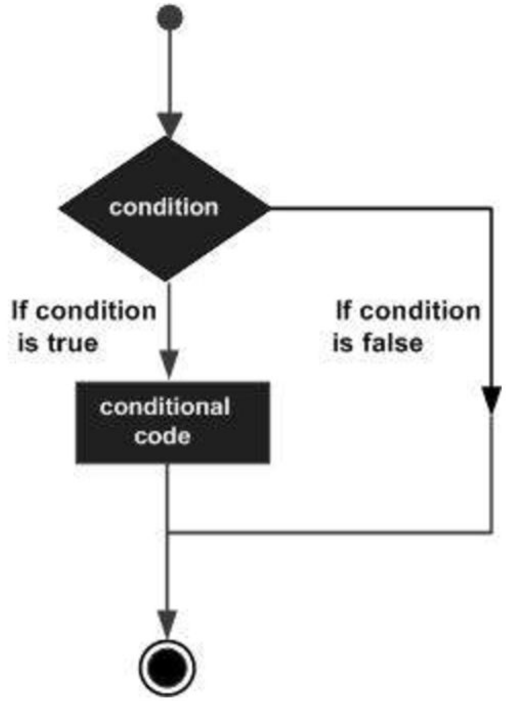
A compound statement consists of one or more ‘clauses.’ A clause consists of a header and a ‘suite.’ Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause.

Compound Statement - If Statement

The if statement is used for **conditional execution**:

if_stmt ::= "if" expression ":" suite
("elif" expression ":" suite)*
["else" ":" suite]

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true; then that suite is executed (and no other part of the if statement is executed or evaluated). If all expressions are false, the suite of the else clause, if present, is executed.



Compound Statement - If Statement

```
>>> x = 10
>>> if x == 10:
...     print('x passes')
... else:
...     print('x fails')
...
x passes
>>> x = 8
>>> if x == 10:
...     print('x passes')
... else:
...     print('x fails')
...
x fails
>>> if x == 10:
...     print('x passes')
... elif x == 8:
...     print('x still passes')
... else:
...     print('x fails')
...
x still passes
```

```
>>> y = 16
>>> if x == 8:
...     if y == 16:
...             print('all good')
...     else:
...             print('y failed')
... else:
...     print('x failed')
...
all good
>>> if x == 8 and y == 16:
...     print('all good')
...
all good
```

Compound Statement - While Loop

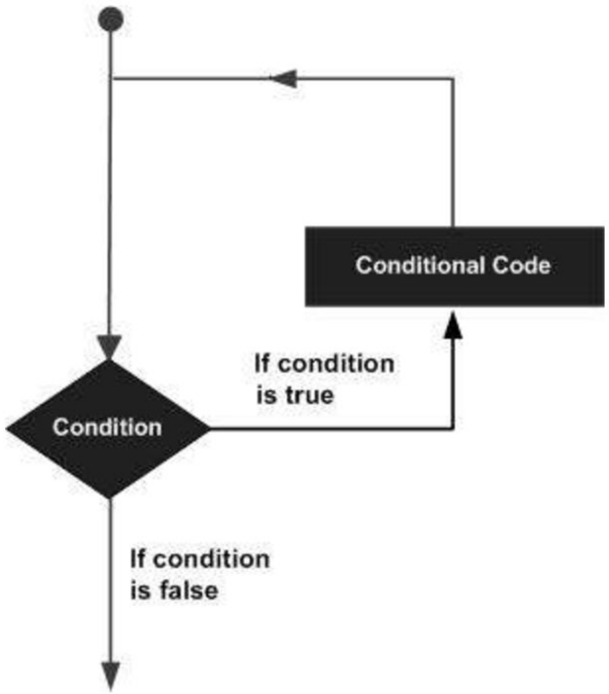
The while statement is used for **repeated execution as long as an expression is true**:

while_stmt ::= "while" expression ":" suite
["else" ":" suite]

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates.

A break statement executed in the first suite terminates the loop without executing the else clause's suite.

A continue statement executed in the first suite skips the rest of the suite and goes back to testing the expression.



Compound Statement - While Loop

```
[>>> i = 0
[>>> while(i < 10):
[...     print(i)
[...     i += 1
[...
0
1
2
3
4
5
6
7
8
9
[>>> i
10
[>>> i = 0
[>>> while(i<10):
[...     print(i)
[...     if i%2 == 0:
[...             i += 2
[...     else:
[...             i -= 1
[...
0
2
4
6
8
>>>
```

Compound Statement - For Loop

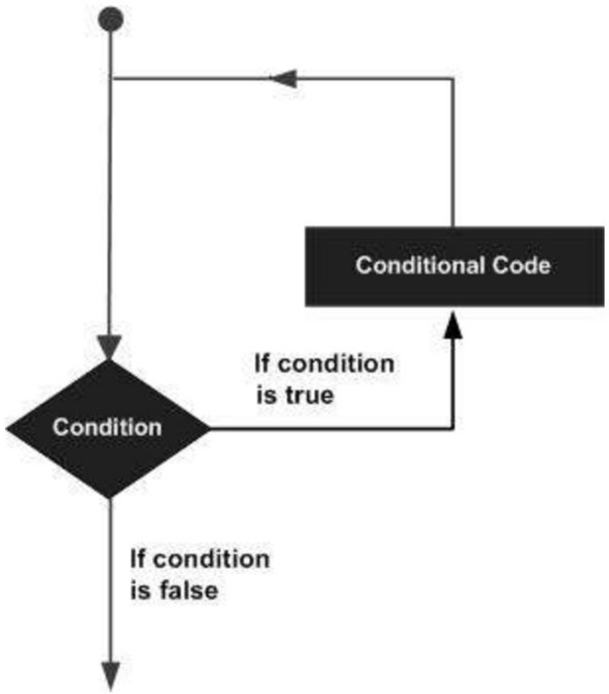
The for statement is **used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object**:

for_stmt ::= "for" target_list "in" expression_list ":" suite
["else" ":" suite]

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a StopIteration exception), the suite in the else clause, if present, is executed, and the loop terminates.

A break statement executed in the first suite terminates the loop without executing the else clause's suite.

A continue statement executed in the first suite skips the rest of the suite and continues with the next item, or with the else clause if there is no next item.



Compound Statement - For Loop

```
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
>>> colors = ['red', 'blue', 'orange', 'brown']
>>> for i in colors:
...     print(i)
...
red
blue
orange
brown
```

Objectives of this Lesson

1. ~~Get Python Running~~
2. ~~The Command-Line, Directories & Files~~
3. ~~What is Python~~
4. ~~Variables~~
5. ~~Input/Output~~
6. ~~Data Types - Numbers: int, float, complex~~
7. ~~Data Types - Text: str~~
8. ~~Data Types - Sequences: lists, tuples, range~~
9. ~~Control Flow & Compound Statements: if, for while~~
10. Simple First Programs

Programming

Algorithm

An algorithm is a step by step process that describes how to solve a problem and/or complete a task, which will always give the correct result.

Pseudo-Code

Algorithms are often expressed using a loosely defined format called pseudo-code, which matches a programming language fairly closely, but leaves out details that could easily be added later by a programmer. Pseudocode doesn't have strict rules about the sorts of commands you can use, but it's halfway between an informal instruction and a specific computer program.

Running a Python Script

So far we've only been working with the interactive Python shell. To get more out of Python we need to create a script, or a series of commands saved in a .py file. Writing a script is a simple as writing the commands that accomplish your algorithm in a .txt file and then changing the file extension from .txt to .py (this is a less than ideal environment to code in, next class we'll look at IDEs for Python - software that makes writing code easier. Check it out on your own in the meantime: Atom, Notepad++, etc). To run your script, in your command-line tool navigate to the directory the script is in and then run the command:

```
$ python3 name_of_your_script.py
```

Programming - Fizz Buzz

Write a program that prints the numbers from 1 to 100.

But for multiples of three print “Fizz” instead of the number

And for the multiples of five print “Buzz”.

For numbers which are multiples of both three and five print “FizzBuzz”."

Programming - Fizz Buzz Solution

```python
# Loop through integers: 1, 2, 3,...,100
for i in range(1, 101):

    # Test Divisibility of i by 3
    if (i % 3 == 0):
        print('Fizz', end = '')         # Suppress the newline w/ end = ''

    # Test Divisibility of i by 5
    if (i % 5 == 0):
        print('Buzz', end = '')

    # Test if i is NOT Divisible by both 3 and 5
    if (i % 3 != 0) and (i % 5 != 0):
        print(i, end = '')

    print()                             # Newline
```

```
slr:Desktop sammishra$ python3 fizzbuzz.py
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
```

This is not the only correct way to code a solution to this problem

Programming - Simple Primality Testing

The simplest primality test is *trial division*:

> Given an input number n, check whether any integer m from 2 to $\sqrt{n}$ evenly divides n (the division leaves no remainder).
>
> If n is divisible by any m then n is composite, otherwise it is prime.

Task: Write a script that asks the user for a number and then uses trial division to test if that number is prime.

Hints: input(prompt) returns a string, before you do any math to this input you must convert this string to a numeric data type, do this with one line of code this way: int(input(prompt)).

pow(num, .5) will give you the square root of num

use round(num) to round num to the nearest integer or int(num) to just drop the decimal.

Keep in mind how range(start,stop,step) works and pay particular attention to how the stop value works.

Programming - Simple Primality Testing Solution

```
num = int(input('Enter a numer to be check for primality: '))
ceiling = round(pow(num, .5)) + 1

for i in range(2, ceiling):
    if (num % i == 0):
        print('{} is composite'.format(num))
        break
else:
        print('{} is prime'.format(num))
```

```
slr:Desktop sammishra$ python3 trialdivision.py
Enter a numer to be check for primality: 17
17 is prime
slr:Desktop sammishra$ python3 trialdivision.py
Enter a numer to be check for primality: 15
15 is composite
```