

Python Programming

Alex Pacheco

LTS Research Computing

What is Python?

- A general-purpose programming language (1980) by Guido van Rossum
- Intuitive and minimal coding
- Dynamically typed
- Automatic memory management
- Interpreted not compiled
- Free (developed under an OSI-approved open-source license) and portable

What can Python do?

- web development (server-side),
- system scripting,
- connecting to database systems and to read and modify files,
- handle big data and perform complex mathematics,
- rapid prototyping, or for production-ready software development.

Why Python?

- works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- has a simple syntax similar to the English language.
- has syntax that allows developers to write programs with fewer lines than some other programming languages.
- runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- can be treated in a procedural way, an object-orientated way or a functional way.
- The most recent major version of Python is Python 3
 - However, Python 2, although not being updated with anything other than security updates, is still quite popular.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes.
- Other programming languages often use curly-brackets for this purpose.

Installing Python

- Many PCs and Macs will have python already installed.
- To check if you have python installed:
 - Open Command Line (cmd.exe) on Windows
 - Open Terminal on Linux or Mac
- and run the following command
 - `python --version`

Installing from Source

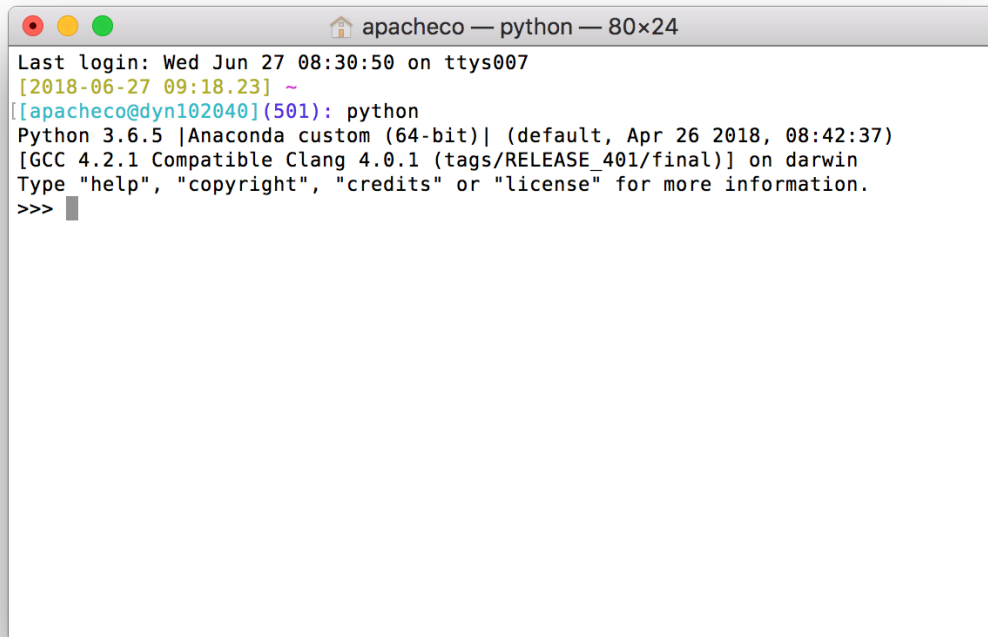
- Python is a free and open source software that can downloaded and installed from <https://www.python.org/downloads/> (<https://www.python.org/downloads/>).
- Latest stable release for Python 3 is 3.7.0
- A large majority of users still use the older version Python 2.
- Python 2 is scheduled for *end-of-life* on Jan 1, 2020 and migrating to Python 3 is strongly recommended.

Anaconda Python Distribution

- Anaconda Python distribution (<https://www.anaconda.com/distribution/>), is the most popular platform for Python
- It provides
 - a convenient install procedure for over 1400 Data Science libraries for Python and R
 - conda to manage your packages, dependencies, and environments
 - anaconda navigator: a desktop portal to install and launch applications and editors including Jupyter, RStudio, Visual Studio Code, and Spyder
- Visit <https://go.lehigh.edu/linux> (<https://go.lehigh.edu/linux>) to use Anaconda (and other Linux software) installed and maintained by the Research Computing group on your local Linux laptop or workstation

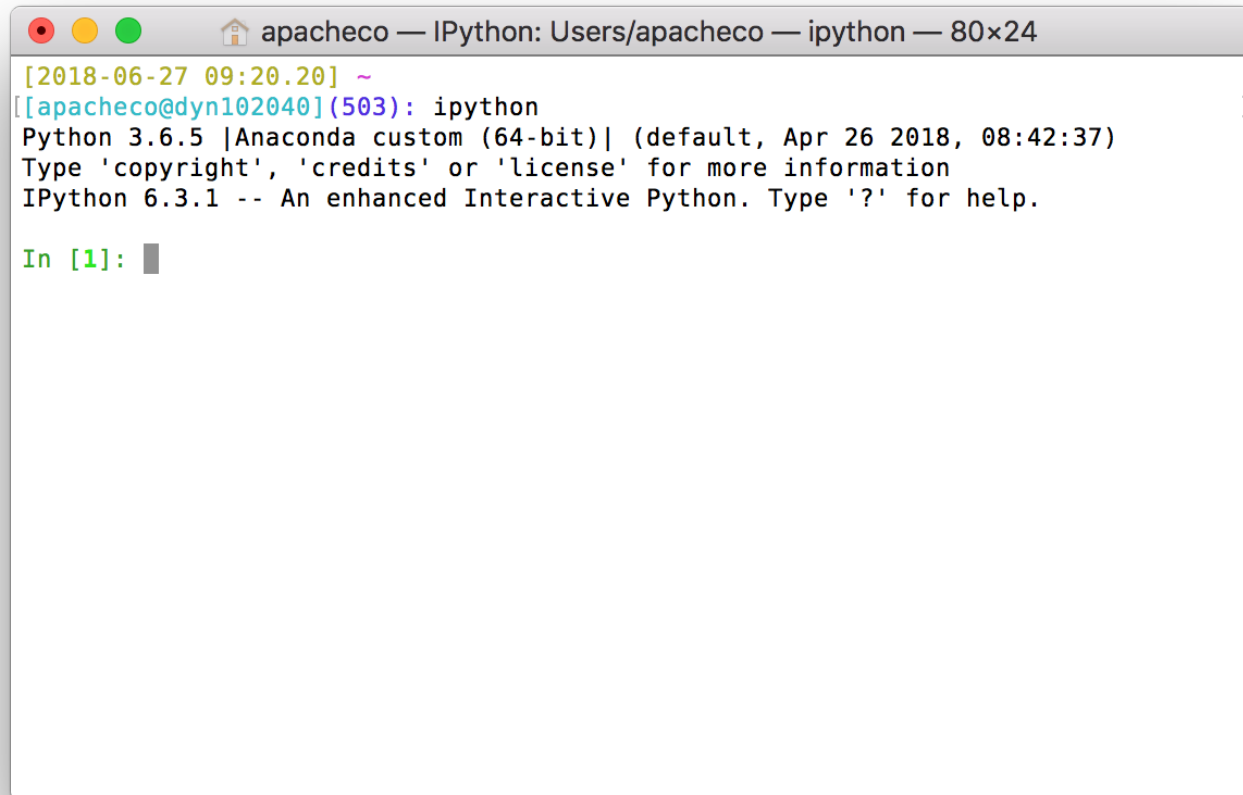
Using Python

- launch python by typing python on the *nix command line or cmd.exe in windows

A screenshot of a terminal window titled 'apacheco — python — 80x24'. The window shows the output of typing 'python' at a shell prompt. The output includes the last login time, the current date and time, the user's name, and the Python version and environment details. The prompt is now '>>>' indicating the Python interpreter is running.

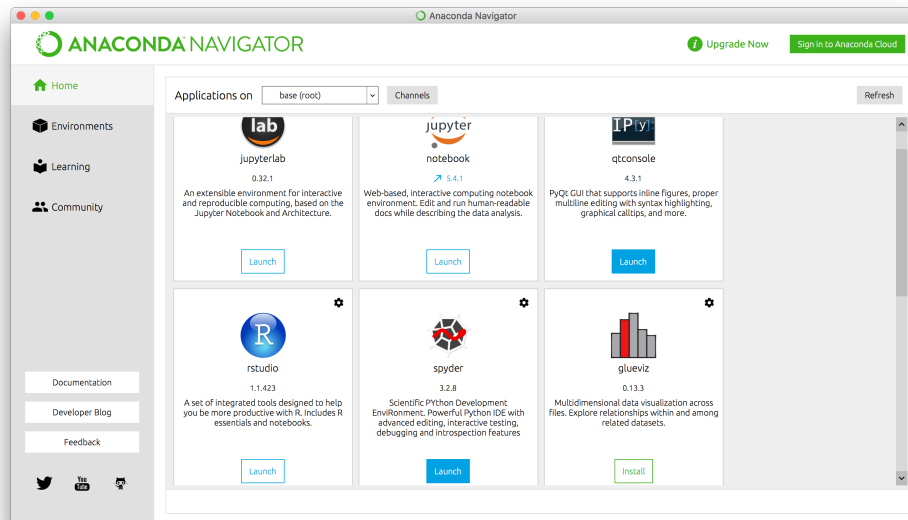
```
apacheco — python — 80x24
Last login: Wed Jun 27 08:30:50 on ttys007
[2018-06-27 09:18.23] ~
[[apacheco@dyn102040](501): python
Python 3.6.5 |Anaconda custom (64-bit)| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- If you have installed Anaconda Python, then you can launch IPython, an enhanced python shell, from the command line

A screenshot of a macOS terminal window. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left, followed by a home icon and the text "apacheco — IPython: Users/apacheco — ipython — 80x24". The terminal content shows a timestamp "[2018-06-27 09:20.20] ~" in yellow, followed by a prompt "[apacheco@dyn102040](503):" in blue and the command "ipython" in black. The output consists of three lines: "Python 3.6.5 |Anaconda custom (64-bit)| (default, Apr 26 2018, 08:42:37)" in black, "Type 'copyright', 'credits' or 'license' for more information" in black, and "IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help." in black. Below this, a new prompt "In [1]:" is shown in green, followed by a black cursor bar.

```
[2018-06-27 09:20.20] ~  
[apacheco@dyn102040](503): ipython  
Python 3.6.5 |Anaconda custom (64-bit)| (default, Apr 26 2018, 08:42:37)  
Type 'copyright', 'credits' or 'license' for more information  
IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help.  
  
In [1]: █
```


- Open Anaconda Navigator to launch
 - Jupyter QtConsole
 - Spyder, an open source IDE for python
 - Jupyter (formerly IPython) notebooks
 - Jupyter Lab (formerly Jupyter Hub)



Which one to use?

- You should choose one that suits your need
 - Python Shell, IPython or QtConsole: for interactive use when you do not want to save your work
 - Spyder or other IDE's such as PyCharm: for writing scripts that you want to save for later reuse
 - Jupyter Lab or Notebooks: for writing scripts and workflows that you can share with others, ideal for reproducible research or data reporting
 - This presentation is written in Jupyter Notebook
 - Provides magic commands like ! and %% to provide access to *nix commands.
 - Use an editor such as vi/vim, emacs, Notepad++ etc to write a python script and execute as a batch script.

Printing to screen

- Python has a built-in function, *print* to write output to standard output or screen

```
In [1]: print("Hello World!")
```

```
Hello World!
```

Reading from screen

- Python has a built-in function, *input* to read input from standard input or screen

```
In [2]: input('What is your name? ')
```

```
What is your name? Alex Pacheco
```

```
Out[2]: 'Alex Pacheco'
```

Your First Python Script

- Create a file, *myscript.py*, with the following content
`print("Hello There!")`
- On the command line, type *python myscript.py* and hit enter

In [3]: `!python myscript.py`

Hello There!

- If you are using Jupyter Notebooks, then bang (!) is used to invoke shell commands
- This is the same as running `python myscript.py` on the command line

Your First Python Script

- On Linux and Mac, add `#!/usr/bin/env python` as the first line
- Convert *myscript.py* to an executable and execute it

```
In [4]: !cat myscript.py
```

```
#!/usr/bin/env python
```

```
print("Hello There!")
```

```
In [5]: %%bash
        chmod +x myscript.py
        ./myscript.py
```

```
Hello There!
```

- If you are using Jupyter Notebooks, then `%%bash` is used to enter a series of bash command
 - If you only need to run one command, then use `!` followed by the command

Python Coding Syntax

- Python uses indentation for blocks, instead of curly braces.
- Both tabs and spaces are supported, but the standard indentation requires standard Python code to use four spaces.

Variables and Types

- A **variable** is a name that refers to a value.
- An **assignment statement** creates new variables and gives them values:

```
In [6]: message = 'And now for something completely different'
        n = 17
        pi = 3.1415926535897931
```

- The first assigns a string to a new variable named message
- the second gives the integer 17 to n
- the third assigns the (approximate) value of π to pi
- To display the value of a variable, you can use a *print* function

```
In [7]: print(message)
```

And now for something completely different

- The type of a variable is the type of the value it refers to.

```
In [8]: type(message)
```

```
Out[8]: str
```

```
In [9]: type(n)
```

```
Out[9]: int
```

```
In [10]: type(pi)
```

```
Out[10]: float
```


Variable names

- Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter.
- The underscore character (_) can appear in a name.
 - It is often used in names with multiple words, such as *my_name*
- If you give a variable an illegal name, you get a syntax error:

```
In [11]: 76trombones = 'big parade'
```

```
File "<ipython-input-11-ee59a172c534>", line 1
    76trombones = 'big parade'
           ^
```

SyntaxError: invalid syntax

```
In [12]: class = 'Advanced Theoretical Zymurgy'
```

```
File "<ipython-input-12-73fc4ce1a15a>", line 1
    class = 'Advanced Theoretical Zymurgy'
      ^
```

SyntaxError: invalid syntax

Reserved Words

- Python has 31 keywords or reserved words that cannot be used for variable names.

and	del	for	is	raise
as	elif	from	lambda	return
assert	else	global	not	try
break	except	if	or	while
class	exec	import	pass	with
continue	finally	in	print	yield
def				

- Use an editor that has syntax highlighting, wherein python functions have a different color
 - See previous slides, variable names are in the normal color i.e. black while reserved keywords, for e.g. *class*, are in green

Statements

- A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.
- When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.
- A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

```
In [13]: print(1)
          x = 2
          print(x)
```

```
1
2
```

Data Types

Python has 5 Basic Data types

1. Numbers
 - A. Integers
 - B. Floating Point Numbers
 - C. Complex Numbers
2. Strings
3. Lists
4. Dictionaries
5. Tuples
6. Boolean

Integers

- There is effectively no limit to how long an integer value can be
- You are constrained by the amount of memory your system

```
In [14]: print(123123123123123123123123123123123123123123123 + 1)
```

```
123123123123123123123123123123123123123123124
```

- Python interprets a sequence of decimal digits without any prefix to be a decimal number:

```
In [15]: print(10)
print(0o10)
print(0b10)
```

```
10
```

```
8
```

```
2
```

- Add a prefix to an integer value to indicate a base other than 10:

Prefix	Interpretation	Base
0b	Binary	2
0o	Octal	8
0x	Hexadecimal	16

- The underlying type of a Python integer, irrespective of the base used to specify it, is called int

```
In [16]: type(10)
```

```
Out[16]: int
```

```
In [17]: type(0o10)
```

```
Out[17]: int
```

```
In [18]: type(0x10)
```

```
Out[18]: int
```

Floating Point Number

- The *float* type in Python designates a floating-point number.
- *float* values are specified with a decimal point.
- Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation

```
In [19]: 4.2
```

```
Out[19]: 4.2
```

```
In [20]: type(0.2)
```

```
Out[20]: float
```

```
In [21]: type(.4e7)
```

```
Out[21]: float
```

```
In [22]: 4.2e-4
```

```
Out[22]: 0.00042
```

Floating-Point Representation

- For 64-bit systems, the maximum value a floating-point number can have is approximately 1.8×10^{308}
- Python will indicate a number greater than that by the string *inf*

```
In [23]: 1.79e308
```

```
Out[23]: 1.79e+308
```

```
In [24]: 1.8e+308
```

```
Out[24]: inf
```


- The closest a nonzero number can be to zero is approximately 5.0×10^{-324}
- Anything closer to zero than that is effectively zero

```
In [25]: 5e-324
```

```
Out[25]: 5e-324
```

```
In [26]: 2e-324
```

```
Out[26]: 0.0
```

- Floating point numbers are represented internally as binary (base-2) fractions
- Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value
- In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems

```
In [27]: 0.5 - 0.4 - 0.1
```

```
Out[27]: -2.7755575615628914e-17
```

- You can convert between an *int* and *float* types using built-in functions

```
In [28]: int(4.2)
```

```
Out[28]: 4
```

```
In [29]: float(15)
```

```
Out[29]: 15.0
```

Complex Numbers

- Complex numbers are specified as *(real part)+(imaginary part)j*. For example:

```
In [30]: a = 2 + 3j  
         type(a)
```

```
Out[30]: complex
```

```
In [31]: print(a.real)  
         print(a.imag)
```

```
2.0  
3.0
```

```
In [32]: a*a
```

```
Out[32]: (-5+12j)
```

```
In [33]: a*a.conjugate()
```

```
Out[33]: (13+0j)
```

- The built-in function *complex* can be used convert to complex type

```
In [34]: complex(3)
```

```
Out[34]: (3+0j)
```

Strings

- Strings are sequences of character data. The string type in Python is called *str*.
- String literals may be delimited using either single or double quotes.
 - All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
In [35]: print("I am a string.")
```

```
I am a string.
```

```
In [36]: type("I am a string.")
```

```
Out[36]: str
```

```
In [37]: print('I am too.')
```

```
I am too.
```

```
In [38]: type('I am too.')
```

```
Out[38]: str
```

- A string in Python can contain as many characters as you wish.
 - The only limit is your machine's memory resources.
- A string can also be empty:

```
In [39]: ''
```

```
Out[39]: ''
```

- If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type.
 - If a string is to contain a single quote, delimit it with double quotes and vice versa

```
In [40]: print("This string contains a single quote (') character.")  
         print('This string contains a double quote (") character.')
```

```
This string contains a single quote (') character.  
This string contains a double quote (") character.
```

- Alternatively, escape the quote character using a backslash

```
In [41]: print('This string contains a single quote (\') character.')  
print("This string contains a double quote (\") character.")
```

This string contains a single quote (') character.

This string contains a double quote (") character.

- Escape sequences

Escape Sequence	Escaped Interpretation
\'	Literal single quote (') character
\"	Literal double quote (") character
\newline	Newline is ignored
\\	Literal backslash () character
\n	ASCII Linefeed (LF) character
\r	ASCII Carriage Return (CR) character
\t	ASCII Horizontal Tab (TAB) character
\v	ASCII Vertical Tab (VT) character

Interactive Python - User Input

- To interact with standard input, the *input* function can be assigned to a string variable
- Modify your *myscript.py* file to ask for your name or anyone's name interactively

```
In [42]: %%bash
cat hello.py
python hello.py

name = input ("What is your name? ")
print("Hello ",name)

What is your name?

Traceback (most recent call last):
  File "hello.py", line 1, in <module>
    name = input ("What is your name? ")
EOFError: EOF when reading a line
```

```
In [43]: # To run example within Jupyter Notebook or IPython or Python Shell
name = input ("What is your name? ")
print("Hello ",name)
```

```
What is your name? Alex Pacheco
Hello  Alex Pacheco
```

- You need to use built-in functions, *int* or *float* to read integer or floating numbers from standard input

Triple Quoted Strings

- Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes.
- Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them.
- This provides a convenient way to create a string with both single and double quotes in it

```
In [44]: print(''This string has a single (') and a double (") quote.'')
```

```
This string has a single (') and a double (") quote.
```

- Because newlines can be included without escaping them, this also allows for multiline strings:

```
In [45]: print("""This is a
string that spans
across several lines""")
```

```
This is a
string that spans
across several lines
```


Boolean

- Objects of Boolean type may have one of two values, *True* or *False*:

```
In [46]: type(True)
```

```
Out[46]: bool
```

```
In [47]: type(False)
```

```
Out[47]: bool
```

Operators and Operands

- Operators are special symbols that represent computations like addition and multiplication.
- The values the operator is applied to are called operands.

Arithmetic Operators

Operator	Meaning	Example
+ (unary)	Unary Positive	+a
+ (binary)	Addition	a + b
- (unary)	Unary Negation	-a
- (binary)	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b
//	Floor Division (also called Integer Divison	a // b
**	Exponentiation	a ** b

```
In [49]: a = 4  
b = 3  
print(+a)  
print(-b)
```

```
4  
-3
```

```
In [50]: print(a + b)  
print(a - b)
```

```
7  
1
```

```
In [51]: print(a * b)  
print(a / b)  
print(3 * a // b)  
print(2 * a % b)  
print(a ** b)
```

```
12  
1.3333333333333333  
4  
2  
64
```

String Operations

- Python strings are immutable i.e. once a string is created it can't be modified

```
In [52]: str1="Hello"
          str2="Hello"
          print(id(str1),id(str2))
```

```
4359750688 4359750688
```

- *str1* and *str2* both refer to the same memory location
- If you modify *str1*, it creates a new object at a different memory location

```
In [53]: str1+=" , Welcome to LTS Seminars"
          print(str1,id(str1),id(str2))
```

```
Hello, Welcome to LTS Seminars 4359947488 4359750688
```

- Every element of a string can be referenced by their index (first index is 0)

```
In [54]: str1[0]
```

```
Out[54]: 'H'
```

- + operator is used to concatenate string and * operator is a repetition operator for string.

```
In [55]: s = "So, you want to learn Python? " * 4  
print(s)
```

So, you want to learn Python? So, you want to learn Python? So, you want to learn Python? So, you want to learn Python?

- You can take subset of string from original string by using [] operator, also known as slicing operator.
- `s[start:end]` will return part of the string starting from index `start` to index `end - 1`

```
In [56]: s[3:15]
```

```
Out[56]: ' you want to'
```

- `start` index and `end` index are optional.
- default value of `start` index is 0
- default value of `end` is the last index of the string

```
In [57]: s[:3]
```

```
Out[57]: 'So,'
```

```
In [58]: s[15:]
```

```
Out[58]: ' learn Python? So, you want to learn Python? So, you want to learn Python? So, you want to learn Python? '
```

```
In [59]: s[:]
```

```
Out[59]: 'So, you want to learn Python? So, you want to learn Python? So, you want to le  
arn Python? So, you want to learn Python? '
```

String functions

- *ord()*: returns the ASCII code of the character.
- *chr()*: function returns character represented by a ASCII number.
- *len()*: returns length of the string
- *max()*: returns character having highest ASCII value
- *min()*: returns character having lowest ASCII value

```
In [60]: len(s)
```

```
Out[60]: 120
```


Comparison Operators

- Can be used for both numbers and strings
- Python compares string lexicographically i.e using ASCII value of the characters

Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal to	a != b
<	Less than	a < b
<=	Less than or equal to	a <= b
>	Greater than	a > b
>=	Greater than or equal to	a >= b

```
In [61]: a = 10  
b = 20  
print(a == b)
```

False

```
In [62]: c = 'Hi'  
d = 'hi'  
print(c == d)  
print( c < d)  
print(ord('H'),ord('h'),ord('i'))
```

False

True

72 104 105

- Consider two strings 'Hi' and 'HI' for comparison.
- The first two characters (H and H) are compared.
- Since 'i' has a greater ASCII value (105) than 'I' with ASCII value (73), 'Hi' is greater than 'HI'

```
In [63]: 'HI' < 'Hi'
```

```
Out[63]: True
```

Logical Operators

Operator	Example	Meaning
not	not x	True if x is False False if x is True (Logically reverses the sense of x)
or	x or y	True if either x or y is True False otherwise
and	x and y	True if both x and y are True False otherwise

```
In [64]: x = 5  
        not x < 10
```

```
Out[64]: False
```

```
In [65]: x < 10 or callable(x)
```

```
Out[65]: True
```

```
In [66]: x < 10 and callable(len)
```

```
Out[66]: True
```

Functions

- A **function** is a named sequence of statements
 - functions are specified by name and a sequence of statements.
 - You "call" the function by name.

```
In [67]: print(x)
```

5

- The name of the function is *print*.
- The expression in parentheses is called the **argument** of the function.
- The result, for this function, is the type of the argument.
- It is common to say that a function "takes" an argument and "returns" a result.
- The result is called the **return value**.

Built-in Type Conversion Functions

Function	Description
ascii()	Returns a string containing a printable representation of an object
bin()	Converts an integer to a binary string
bool()	Converts an argument to a Boolean value
callable()	Returns whether the object is callable (i.e., some kind of function)
chr()	Returns string representation of character given by integer argument
complex()	Returns a complex number constructed from arguments
float()	Returns a floating-point object constructed from a number or string
hex()	Converts an integer to a hexadecimal string
int()	Returns an integer object constructed from a number or string
oct()	Converts an integer to an octal string
ord()	Returns integer representation of a character
repr()	Returns a string containing a printable representation of an object
str()	Returns a string version of an object
type()	Returns the type of an object or creates a new type object

Math

- Python has a math module that provides most of the familiar mathematical functions

Function	Description
abs()	Returns absolute value of a number
divmod()	Returns quotient and remainder of integer division
max()	Returns the largest of the given arguments or items in an iterable
min()	Returns the smallest of the given arguments or items in an iterable
pow()	Raises a number to a power
round()	Rounds a floating-point value
sum()	Sums the items of an iterable

- A **module** is a file that contains a collection of related functions
- Before we can use the module, we have to import it

```
In [68]: import math
```

- This statement creates a **module object** named *math*
- If you print the module object, you get some information about it

```
In [69]: print(math)
```

```
<module 'math' from '/Users/apacheco/anaconda3/lib/python3.6/lib-dynload/math.cpython-36m-darwin.so'>
```

- The module object contains the functions and variables defined in the module
- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period)
- This format is called **dot notation**

```
In [70]: degrees = 45  
radians = degrees / 360.0 * 2 * math.pi  
math.sin(radians)
```

```
Out[70]: 0.7071067811865475
```

Iterables and Iterators

Function	Description
all()	Returns True if all elements of an iterable are true
any()	Returns True if any elements of an iterable are true
enumerate()	Returns a list of tuples containing indices and values from an iterable
filter()	Filters elements from an iterable
iter()	Returns an iterator object
len()	Returns the length of an object
map()	Applies a function to every item of an iterable
next()	Retrieves the next item from an iterator
range()	Generates a range of integer values
reversed()	Returns a reverse iterator
slice()	Returns a slice object
sorted()	Returns a sorted list from an iterable
zip()	Creates an iterator that aggregates elements from iterables

User Defined Functions

- Python allows programmers to define their OWN **function**
- A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called.

Why create your own functions?

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

```
In [71]: def celcius_to_fahrenheit(tempc):  
        tempf = 9.0 / 5.0 * tempc + 32.0  
        return tempf
```

```
In [72]: tempc = float(input('Enter Temperature in Celcius: '))  
  
        print("%6.2f C = %6.2f F" % (tempc, celcius_to_fahrenheit(tempc)))
```

```
Enter Temperature in Celcius: 25  
25.00 C = 77.00 F
```

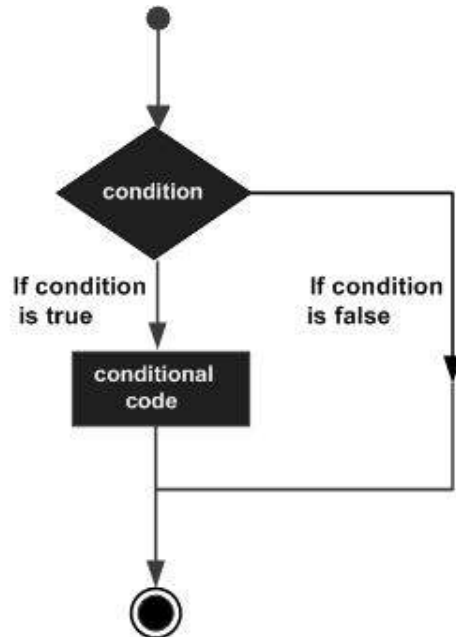
- When you create a variable inside a function, it is **local**, which means that it only exists inside the function.
 - e.g. *tempf* is local within the *celcius_to_fahrenheit* function and does not exist outside the scope of the function

```
In [73]: print(tempf)
```

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-73-8f86e1a6cd78> in <module>()  
----> 1 print(tempf)  
  
NameError: name 'tempf' is not defined
```

Conditional Execution

- **Conditional Statements** gives the programmer an ability to check conditions and change the behavior of the program accordingly.
- The simplest form is the if statement:



Syntax:

```
if condition:  
    statements
```

- The boolean expression after the *if* statement is called the **condition**.
 - If it is true, then the indented statement gets executed.
 - If not, nothing happens.
- *if* statements have the same structure as function definitions:
 - a header followed by an indented block.
 - Statements like this are called **compound statements**.

- There is no limit on the number of statements that can appear in the body, but there has to be at least one.
- Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet).
- In that case, you can use the *pass* statement, which does nothing.

```
In [74]: if x < 0:  
         pass
```

- There may be a situation where you want to execute a series of statements if the *condition* is false
- Python provides an `if ... else ...` conditional
- Syntax

```
if condition:
    statements_1
else:
    statements_2
```

```
In [75]: if x % 2 == 0:
          print('x is even')
          else:
          print('x is odd')
```

```
x is odd
```

- If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect.
- If the condition is false, the second set of statements is executed.
- Since the condition must be true or false, exactly one of the alternatives will be executed. - The alternatives are called **branches**, because they are branches in the flow of execution.

- Sometimes there are more than two possibilities and we need more than two branches.
- Python provides a `if ... elif ... else` conditional
- Syntax

```
if condition1:
    statements_1
elif condition2:
    statements_2
else
    statements_3
```

```
In [76]: y = 10
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

x is less than y

- *elif* is an abbreviation of “else if.”
- Again, exactly one branch will be executed.
- There is no limit on the number of *elif* statements.
- If there is an *else* clause, it has to be at the end, but there doesn’t have to be one.

```
In [77]: choice='d'
         if choice == 'a':
             print('choice is a')
         elif choice == 'b':
             print('choice is b')
         elif choice == 'c':
             print('choice is c')
```

- Each condition is checked in order.
- If the first is false, the next is checked, and so on.
- If one of them is true, the corresponding branch executes, and the statement ends.
- Even if more than one condition is true, only the first true branch executes.

- Conditional can also be nested within another.

```
In [78]: if x == y:
        print('x and y are equal')
        else:
            if x < y:
                print('x is less than y')
            else:
                print('x is greater than y')
```

x is less than y

- The outer conditional contains two branches.
- The first branch contains a simple statement.
- The second branch contains another *if* statement, which has two branches of its own.
- Those two branches are both simple statements, although they could have been conditional statements as well.

- Logical operators often provide a way to simplify nested conditional statements.

```
In [79]: if 0 < x:
        if x < 10:
            print('x is a positive single-digit number.')
```

x is a positive single-digit number.

- The *print* statement is executed only if we make it past both conditionals
- we can get the same effect with the *and* operator

```
In [80]: if 0 < x and x < 10:
        print('x is a positive single-digit number.')
```

x is a positive single-digit number.

Control Statements

Python provides three control statements that can be used within conditionals and loops

1. **break:** Terminates the loop statement and transfers execution to the statement immediately following the loop
2. **continue:** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3. **pass:** The pass statement is used when a statement is required syntactically but you do not want any command or code to execute.

Recursion

- Python functions can call itself recursively

```
In [81]: def factorial(n):  
        if n < 1:  
            return 1  
        else:  
            return n*factorial(n-1)
```

```
In [82]: factorial(5)
```

```
Out[82]: 120
```

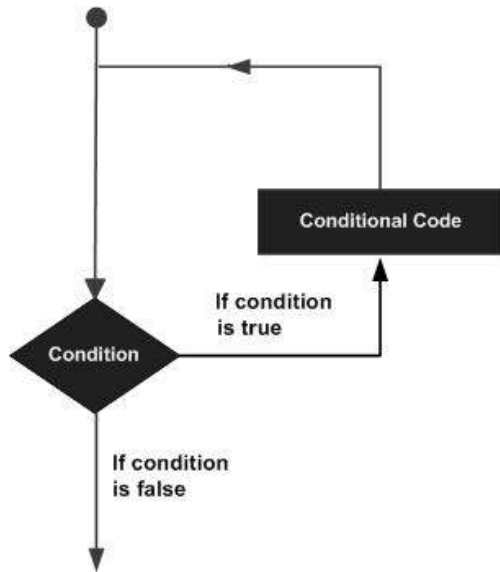
```
In [83]: def double_fact(n):  
        if n < 2:  
            return 1  
        else:  
            return n * double_fact(n - 2)
```

```
In [84]: double_fact(10)
```

```
Out[84]: 3840
```

Loops

- There may be a situation when you need to execute a block of code a number of times.
- A loop statement allows us to execute a statement or group of statements multiple times.



for loops

- The *for* statement has the ability to iterate over the items of any sequence, such as a list or a string
- If a sequence contains an expression list, it is evaluated first.
- Then, the first item in the sequence is assigned to the iterating variable `iterating_var`.
- Next, the statements block is executed.
- Each item in the list is assigned to `iterating_var`, and the statement(s) block is executed until the entire sequence is exhausted.

```
for iterating_var in sequence:  
    statements(s)
```

```
In [85]: for letter in 'Hola':  
        print('Current Letter :', letter)
```

```
Current Letter : H  
Current Letter : o  
Current Letter : l  
Current Letter : a
```

```
In [86]: fruits = ['banana', 'apple', 'mango']  
        for fruit in fruits:  
            print ('Current fruit :', fruit)
```

```
Current fruit : banana  
Current fruit : apple  
Current fruit : mango
```

- An alternative way of iterating through each item is by index offset into the sequence itself

```
In [87]: fruits = ['banana', 'apple', 'mango']  
        for index in range(len(fruits)):  
            print ('Current fruit :', fruits[index])
```

```
Current fruit : banana  
Current fruit : apple  
Current fruit : mango
```


range function

- The built-in function *range()* iterates over a sequence of numbers.

```
In [88]: range(5)
```

```
Out[88]: range(0, 5)
```

```
In [89]: list(range(5))
```

```
Out[89]: [0, 1, 2, 3, 4]
```

```
In [90]: for var in list(range(5)):  
         print(var)
```

```
0  
1  
2  
3  
4
```

while loop

- A *while* loop statement repeatedly executes a target statement as long as a given condition is true.
- Here, **statement(s)** may be a single statement or a block of statements with uniform indent.
- The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.

```
while expression:  
    statement(s)
```

```
In [91]: number = int(input('Enter any integer: '))

fact = count = 1
while (count <= number ):
    fact = count * fact
    count += 1
print('Factorial of %d is %d' % (number, fact))
```

Enter any integer: 15

Factorial of 15 is 1307674368000

infinite loop

- A loop becomes infinite loop if a condition never becomes FALSE

```
number = int(input('Enter any integer: '))  
fact = count = 1  
while (count <= number ):  
    fact = count * fact  
print('Factorial of %d is %d' % (number, fact))
```

Using else Statement with Loops

- Python supports having an *else* statement associated with a loop statement.
- If the *else* statement is used with a *for* loop, the *else* block is executed only if the *for* loop terminates normally (and not by encountering break statement).
- If the *else* statement is used with a *while* loop, then the *else* statement is executed when the condition becomes false.

```
In [92]: numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
    if num%2 == 0:
        print ('the list contains an even number')
        break
    else:
        print ('the list does not contain even number')
```

the list does not contain even number

```
In [93]: count = 0
while count < 5:
    print (count, " is less than 5")
    count = count + 1
else:
    print (count, " is not less than 5")
```

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

Lists

- Like a string, a *list* is a sequence of values.
- In a string, the values are characters; in a list, they can be any type.
- The values in list are called **elements** or sometimes items.

```
In [94]: a = [10, 20, 30, 40]  
         print(a)
```

```
[10, 20, 30, 40]
```

- *lists* can be nested.

```
In [95]: b = ['spam', 2.0, 5, [10, 20]]  
         print(b)
```

```
['spam', 2.0, 5, [10, 20]]
```

- An empty list i.e. list with no elements is created with empty brackets [].

```
In [96]: c=[]  
         print(c)
```

```
[]
```

- Lists are mutable i.e. they can be modified after creation

```
In [97]: numbers = [17, 123]  
         print(numbers)
```

```
[17, 123]
```

```
In [98]: numbers[0] = 5  
         print(numbers)
```

```
[5, 123]
```


- A list can be traversed using a *for* loop

```
In [99]: for i in numbers:  
        print(i)
```

```
5  
123
```

- The + operator concatenates lists:

```
In [100]: a = [1, 2, 3]  
         b = [4, 5, 6]  
         c = a + b  
         print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

- You can reference a section of the list using a slice operator

```
In [101]: c[2:5]
```

```
Out[101]: [3, 4, 5]
```

list operations

- *append* adds a new element to the end of the list

```
In [102]: t1 = ['a', 'b', 'c']  
          t1.append('d')  
          print(t1)
```

```
['a', 'b', 'c', 'd']
```

- *extend* takes a list as an argument and appends all of the elements

```
In [103]: t2 = ['e', 'f']  
          t1.extend(t2)  
          print(t1)
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

- *sort* arranges the elements of the list from low to high:

```
In [104]: t = ['d', 'c', 'e', 'b', 'a']  
          t.sort()  
          print(t)
```

```
['a', 'b', 'c', 'd', 'e']
```

deleting list elements

- *pop* modifies the list and returns the element that was removed.
- If you don't provide an index, it deletes and returns the last element

```
In [105]: t = ['a', 'b', 'c']  
          x = t.pop(1)  
          print(t)
```

```
['a', 'c']
```

```
In [106]: print(x)
```

```
b
```

- Use *del* if you do not need the removed value

```
In [107]: t = ['a', 'b', 'c']  
          del t[1]  
          print(t)  
  
          ['a', 'c']
```

- If you know the element you want to remove (but not the index), you can use *remove*

```
In [108]: t = ['a', 'b', 'c', 'd', 'e', 'f']  
          t.remove('b')  
          print(t)  
  
          ['a', 'c', 'd', 'e', 'f']
```

- To remove more than one element, you can use *del* with a slice index

```
In [109]: t = ['a', 'b', 'c', 'd', 'e', 'f']  
          del t[1:5]  
          print(t)  
  
          ['a', 'f']
```

lists and strings

- A string is a sequence of characters and a list is a sequence of values
- list of characters is not the same as a string.
- To convert from a string to a list of characters, you can use *list*

```
In [110]: s = 'spam'  
          t = list(s)  
          print(t)
```

```
['s', 'p', 'a', 'm']
```

- The *list* function breaks a string into individual letters.
- If you want to break a string into words, you can use the *split* method

```
In [111]: s = 'pinning for the fjords'  
          t = s.split()  
          print(t)
```

```
['pinning', 'for', 'the', 'fjords']
```

- An optional argument called a **delimiter** specifies which characters to use as word boundaries

```
In [112]: s = 'spam-spam-spam'
          delimiter = '-'
          s.split(delimiter)
```

```
Out[112]: ['spam', 'spam', 'spam']
```

- *join* is the inverse of *split*.
- It takes a list of strings and concatenates the elements.
- *join* is a string method, so you have to invoke it on the delimiter and pass the list as a parameter

```
In [113]: t = ['pinning', 'for', 'the', 'fjords']
          delimiter = ':'
          delimiter.join(t)
```

```
Out[113]: 'pinning:for:the:fjords'
```

Dictionaries

- A *dictionary* is a mapping between a set of indices (which are called **keys**) and a set of **values**.
- Each key maps to a value.
- The association of a key and a value is called a **key-value pair**
- The function *dict* creates a new dictionary with no items

```
In [114]: eng2sp = dict()  
          print(eng2sp)
```

```
{}
```

- To add items to the dictionary, you can use square brackets

```
In [115]: eng2sp['one'] = 'uno'  
          print(eng2sp)
```

```
{'one': 'uno'}
```

- You can update a dictionary by adding a new entry or a key-value pair

```
In [116]: eng2sp['two'] = 'dos'  
print(eng2sp)
```

```
{'one': 'uno', 'two': 'dos'}
```

- You can create a dictionary as follows

```
In [117]: eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}  
print(eng2sp)
```

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

- the order of items in a dictionary is unpredictable
- use *keys* to look up the corresponding *value*


```
In [118]: print(eng2sp['three'])
```

```
tres
```

- To delete entries in a dictionary, use *del*

```
In [119]: del eng2sp['two']  
print(eng2sp)
```

```
{'one': 'uno', 'three': 'tres'}
```

- The *clear()* function is used to remove all elements of the dictionary

```
In [120]: eng2sp.clear()  
print(eng2sp)
```

```
{}
```

- The *len* function returns the number of key-value pairs

```
In [121]: eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}  
len(eng2sp)
```

```
Out[121]: 3
```

- You can loop through keys or values by using the *keys* and *values* functions

```
In [122]: for keys in eng2sp.keys():  
           print(keys)
```

```
one  
two  
three
```

```
In [123]: for vals in eng2sp.values():  
           print(vals)
```

```
uno  
dos  
tres
```

Tuples

- A *tuple* is a sequence of values
- The values can be any type, and they are indexed by integers.

```
In [124]: t = ('a', 'b', 'c', 'd', 'e')
```

- To create a tuple with a single element, you have to include the final comma

```
In [125]: t1 = ('a',)  
type(t1)
```

```
Out[125]: tuple
```

```
In [126]: t2 = ('a')  
type(t2)
```

```
Out[126]: str
```

- Another way to create a tuple is the built-in function *tuple*.
- With no argument, it creates an empty tuple

```
In [127]: t = tuple()
          print(t)
```

```
()
```

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence

```
In [128]: t = tuple('lupins')
          print(t)
```

```
('l', 'u', 'p', 'i', 'n', 's')
```

- Most list operators also work on tuples. The bracket operator indexes an element

```
In [129]: t = ('a', 'b', 'c', 'd', 'e')
          print(t[1])
```

```
b
```

- slice operator selects a range of elements

```
In [130]: print(t[1:],id(t))
```

```
('b', 'c', 'd', 'e') 4359797136
```

- Unlike *lists*, *tuples* are immutable

```
In [131]: t[0] = 'A'
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-131-7e674cdf20e6> in <module>()  
----> 1 t[0] = 'A'
```

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

- You can't modify the elements of a tuple, but you can replace one tuple with another

```
In [132]: t = ('A',) + t[1:]  
print(t,id(t))
```

```
('A', 'b', 'c', 'd', 'e') 4359796960
```

Assignment

- It is often useful to swap the values of two variables using a cumbersome procedure

```
In [133]: a = 1  
          b = 2
```

```
In [134]: temp = a  
          a = b  
          b = temp  
          print(a,b)
```

```
2 1
```

- *tuple assignment* is more elegant

```
In [135]: a,b = b,a  
          print(a,b)
```

```
1 2
```

- The number of variables on the left and the number of values on the right have to be the same

```
In [136]: a, b = 1,2,3
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-136-6b644e00ae5e> in <module>()  
----> 1 a, b = 1,2,3
```

```
ValueError: too many values to unpack (expected 2)
```

- Tuples can be used to return multiple values from a function

```
In [137]: quot, rem = divmod(7, 3)  
          print(quot)
```

```
2
```

```
In [138]: print(rem)
```

```
1
```

File Handling

- To read/write a file, you have to open it with an appropriate mode as the second parameter

`open(filename, mode)`

mode	description
r	Opens a file for reading only, default mode
w	Opens a file for writing only
a	Opens a file for appending only. File pointer is at end of file
rb	Opens a file for reading only in binary
wb	Opens a file for writing only in binary


```
In [139]: fout = open('output.txt', 'w')  
          print(fout)
```

```
<_io.TextIOWrapper name='output.txt' mode='w' encoding='UTF-8'>
```

- If the file already exists, opening it in write mode clears out the old data.
- If the file doesn't exist, a new one is created.
- The *write* method puts data into the file.

```
In [140]: line1 = "This here's the wattle,\n"  
          fout.write(line1)
```

```
Out[140]: 24
```

```
In [141]: line2 = "the emblem of our land.\n"  
          fout.write(line2)
```

```
Out[141]: 24
```

- When you are done writing, you have to close the file.

```
In [142]: fout.close()  
!cat output.txt
```

```
This here's the wattle,  
the emblem of our land.
```

- To read data back from the file you need one of these three methods

Method	Description
read([number])	Return specified number of characters from the file. if omitted it will read the entire contents of the file.
readline()	Return the next line of the file.
readlines()	Read all the lines as a list of strings in the file

- Reading all the data at once.

```
In [143]: f = open('myscript.py', 'r')  
f.read()
```

```
Out[143]: '#!/usr/bin/env python\n\nprint("Hello There!")\n\n'
```

```
In [144]: f.close()
```

- Reading all lines as an array

```
In [145]: f = open('myscript.py', 'r')  
f.readlines()
```

```
Out[145]: ['#!/usr/bin/env python\n', '\n', 'print("Hello There!")\n', '\n']
```

```
In [146]: f.close()
```

- Reading only one line.

```
In [147]: f = open('myscript.py', 'r')  
          f.readline()
```

```
Out[147]: '#!/usr/bin/env python\n'
```

```
In [148]: f.close()
```

- You can iterate through the file using file pointer.

```
In [149]: f = open('myscript.py', 'r')  
          for line in f:  
              print(line)  
          f.close()
```

```
#!/usr/bin/env python
```

```
print("Hello There!")
```

Formatted output

- The argument of write has to be a string
- convert other values to strings using *str*

```
In [150]: f = open('output.txt', 'w')
          x = 52
          f.write(str(x))
          f.close()
          !cat output.txt
```

52

- An alternative is to use the **format operator**, %
- The first operand is the **format string**, and the second operand is a **tuple of expressions**.

```
In [151]: tempc = float(input('Enter Temperature in Celcius: '))

          print("%6.2f C = %6.2f F" % (tempc, celcius_to_fahrenheit(tempc)))
```

```
Enter Temperature in Celcius: 30
30.00 C = 86.00 F
```

- The general syntax for print function is `print(format string with placeholder % (variables))`
- The general syntax for a format placeholder is `%[flags][width][.precision]type`

type	data type
s	strings
f or F	floating point numbers
d or i	integers
e or E	Floating point exponential format
g or G	same as e or E if exponent is greater than -4, f or F otherwise

- If you try to open a file that doesn't exist, you get an IOError:

```
In [152]: fin = open('bad_file')
```

```
-----  
FileNotFoundError                                Traceback (most recent call last)  
<ipython-input-152-a7d7d7ad396b> in <module>()  
----> 1 fin = open('bad_file')  
  
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
```

- If you don't have permission to access a file

```
In [153]: fout = open('/etc/passwd', 'w')
```

```
-----  
PermissionError                                Traceback (most recent call last)  
<ipython-input-153-8a9adb191927> in <module>()  
----> 1 fout = open('/etc/passwd', 'w')  
  
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

- or if you try to open a directory for reading

```
In [154]: fin = open('/home')
```

```
-----  
IsADirectoryError                                Traceback (most recent call last)  
<ipython-input-154-a2032f82d461> in <module>()  
----> 1 fin = open('/home')  
  
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

- Python provides statements, *try* and *except* to allow programmers to gracefully quit the program

```
In [155]: try:  
          fin = open('bad.txt')  
          for line in fin:  
              print(line)  
          fin.close()  
except:  
    print('Something went wrong.')
```

```
Something went wrong.
```


In [156]: `!cat test1.py`

```
fin = open('bad.txt')
for line in fin:
    print(line)
fin.close()

print('Hello World!')
```

In [157]: `!python test1.py`

```
Traceback (most recent call last):
  File "test1.py", line 1, in <module>
    fin = open('bad.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'bad.txt'
```

In [158]: `!cat test2.py`

```
try:
    fin = open('bad.txt')
    for line in fin:
        print(line)
    fin.close()
except:
    print('Something went wrong.')

print('Hello World!')
```

In [159]: `!python test2.py`

```
Something went wrong.
Hello World!
```

NumPy

- NumPy is the fundamental package for scientific computing with Python.
- It contains among other things
 - a powerful N-dimensional array object
 - sophisticated (broadcasting) functions
 - tools for integrating C/C++ and Fortran code
 - useful linear algebra, Fourier transform, and random number capabilities
- NumPy can also be used as an efficient multi-dimensional container of generic data.
- Numpy arrays are a great alternative to Python Lists

- NumPy's main object is the homogeneous multidimensional array.
- NumPy's array class is called *ndarray*. It is also known by the alias *array*
- The more important attributes of an *ndarray* object are:
 - `ndarray.ndim`: the number of axes (dimensions) of the array.
 - `ndarray.shape`: the dimensions of the array.
 - `ndarray.size`: the total number of elements of the array.
 - `ndarray.dtype`: an object describing the type of the elements in the array.
 - `ndarray.itemsize`: the size in bytes of each element of the array.
 - `ndarray.data`: the buffer containing the actual elements of the array.

```
In [160]: import numpy as np
a = np.arange(15).reshape(3, 5)
a
```

```
Out[160]: array([[ 0,  1,  2,  3,  4],
                 [ 5,  6,  7,  8,  9],
                 [10, 11, 12, 13, 14]])
```

```
In [161]: print("array dimensions: ", a.shape)
print("number of dimensions: ", a.ndim)
print("element types: ", a.dtype.name)
print("size of elements: ", a.itemsize)
print("total number of elements: ",a.size)
print("type of object:",type(a))
```

```
array dimensions:  (3, 5)
number of dimensions:  2
element types:  int64
size of elements:  8
total number of elements:  15
type of object: <class 'numpy.ndarray'>
```

Numpy Arrays

- create an array from a regular Python *list* or *tuple* using the *array* function

```
In [162]: a = np.array([2,3,4])  
          print(a, a.dtype)
```

```
[2 3 4] int64
```

- *array* transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays

```
In [163]: b = np.array([(1.2, 3.5, 5.1), (4.1, 6.1, 0.5)])  
          print(b, b.dtype)
```

```
[[1.2 3.5 5.1]  
 [4.1 6.1 0.5]] float64
```

- The type of the array can also be explicitly specified at creation time

```
In [164]: c = np.array( [ [1,2], [3,4] ], dtype=complex )  
c
```

```
Out[164]: array([[1.+0.j, 2.+0.j],  
                [3.+0.j, 4.+0.j]])
```

- The function *zeros* creates an array full of zeros,
- the function *ones* creates an array full of ones, and
- the function *empty* creates an array whose initial content is random and depends on the state of the memory.

```
In [165]: print('Zeros: ', np.zeros( (3,4) ))  
print('Ones', np.ones( (2,4), dtype=np.float64 ))  
print('Empty', np.empty( (2,3) ))
```

```
Zeros: [[0. 0. 0. 0.]  
        [0. 0. 0. 0.]  
        [0. 0. 0. 0.]  
Ones [[1. 1. 1. 1.]  
      [1. 1. 1. 1.]  
Empty [[1.2 3.5 5.1]  
       [4.1 6.1 0.5]]
```

- NumPy provides a function analogous to range that returns arrays instead of lists.

```
In [166]: np.arange( 10, 30, 5 )
```

```
Out[166]: array([10, 15, 20, 25])
```

- Arithmetic operators on arrays apply elementwise

```
In [167]: a = np.array( [20,30,40,50] )  
b = np.arange( 4 )  
print('a: ', a)  
print('b:', b)  
print('a-b:', a-b)  
print('B**2', b**2)  
print('10*sin(a): ', 10*np.sin(a))  
print('Which elements of a < 35:', a<35)  
print('Elements of a < 35: ',a[a<35])
```

```
a: [20 30 40 50]  
b: [0 1 2 3]  
a-b: [20 29 38 47]  
B**2 [0 1 4 9]  
10*sin(a): [ 9.12945251 -9.88031624  7.4511316  -2.62374854]  
Which elements of a < 35: [ True  True False False]  
Elements of a < 35: [20 30]
```



```
In [168]: A = np.array( [[1,1], [0,1]] )
          B = np.array( [[2,0], [3,4]] )
          print('A = ', A)
          print('B = ', B)
          print('A*B = ', A*B)
          print('A . B = ', A.dot(B))
          print('Numpy A. B = ', np.dot(A, B))
```

```
A =  [[1 1]
      [0 1]]
B =  [[2 0]
      [3 4]]
A*B =  [[2 0]
        [0 4]]
A . B =  [[5 4]
          [3 4]]
Numpy A. B =  [[5 4]
               [3 4]]
```

Visualization

- Matplotlib is the most popular visualization tool used in Python
- Other visualization tools commonly used are bokeh, seaborn and plot.ly

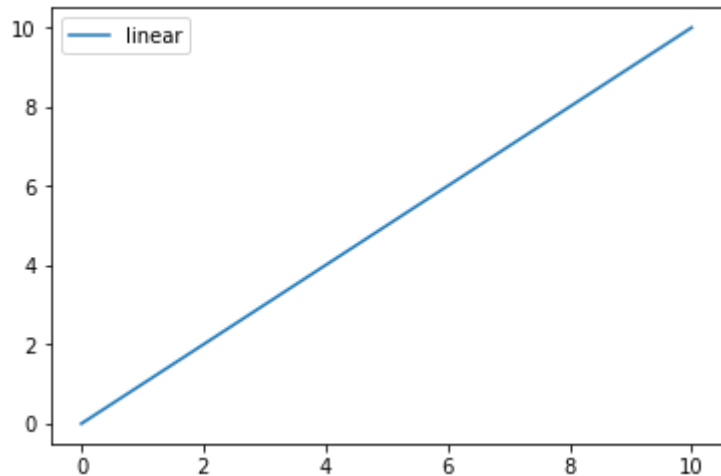
```
In [169]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

# Prepare the data
x = np.linspace(0, 10, 100)

# Plot the data
plt.plot(x, x, label='linear')

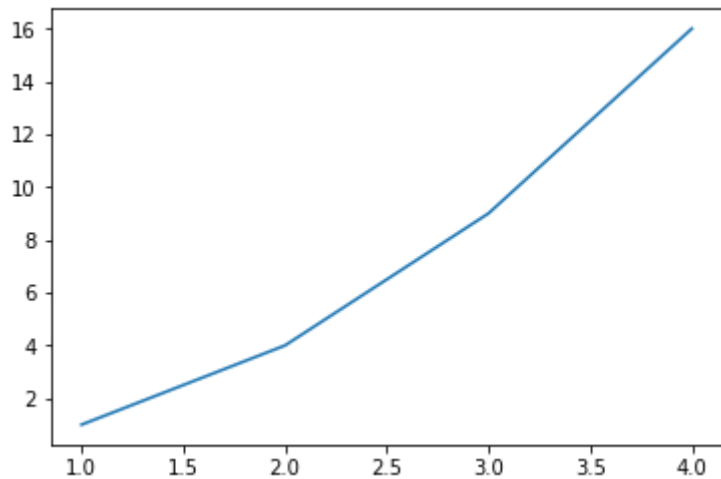
# Add a legend
plt.legend()

# Show the plot
plt.show()
```

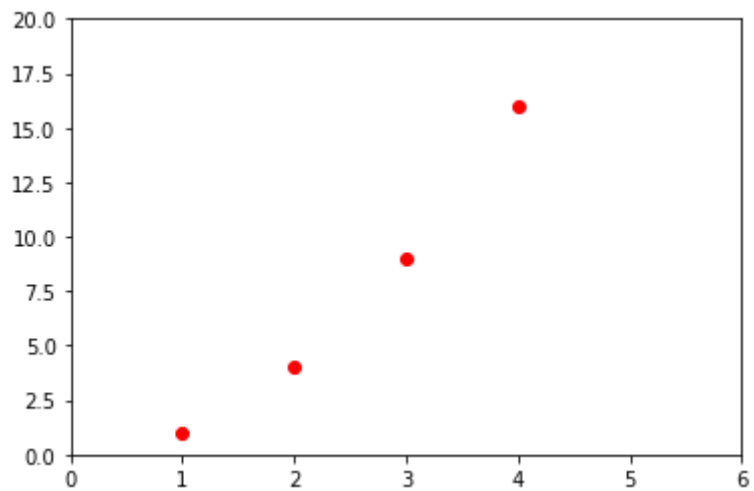


```
In [170]: plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

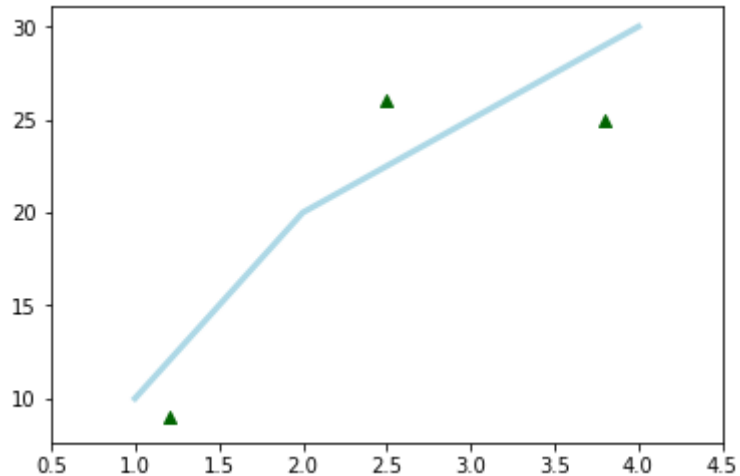
```
Out[170]: [<matplotlib.lines.Line2D at 0x10b24b400>]
```



```
In [171]: plt.plot([1,2,3,4], [1,4,9,16], 'ro')  
plt.axis([0, 6, 0, 20])  
plt.show()
```

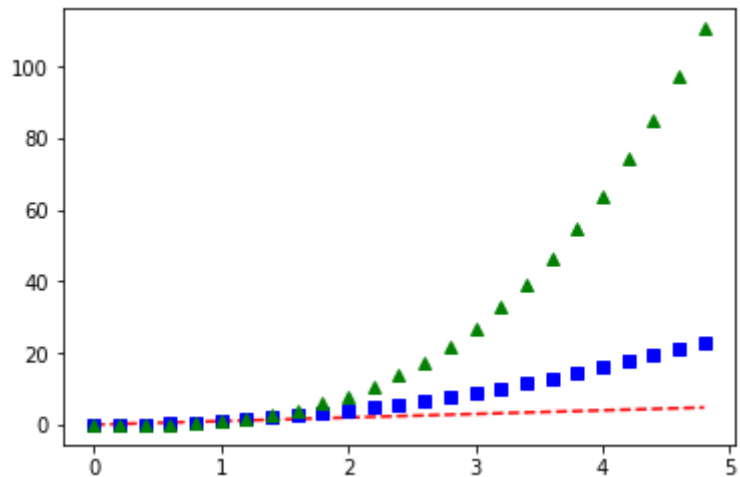


```
In [172]: fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1, 2, 3, 4], [10, 20, 25, 30], color='lightblue', linewidth=3)
ax.scatter([0.3, 3.8, 1.2, 2.5], [11, 25, 9, 26], color='darkgreen', marker='^')
ax.set_xlim(0.5, 4.5)
plt.show()
```

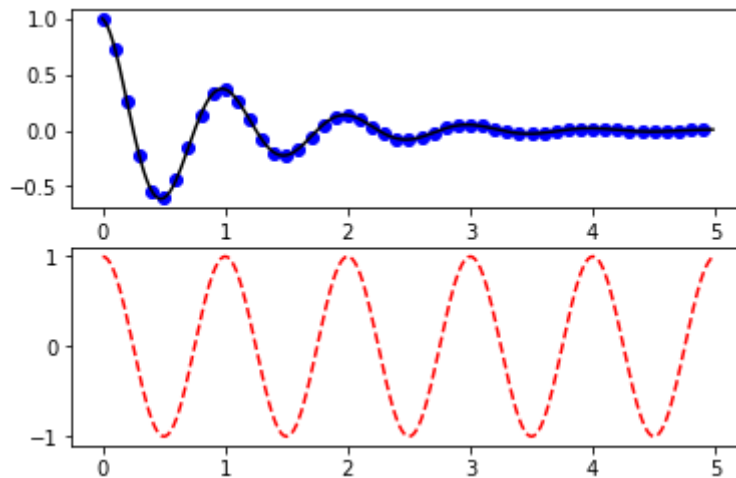


```
In [173]: t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



```
In [174]: def f(t):  
            return np.exp(-t) * np.cos(2*np.pi*t)  
  
t1 = np.arange(0.0, 5.0, 0.1)  
t2 = np.arange(0.0, 5.0, 0.02)  
  
plt.figure(1)  
plt.subplot(211)  
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')  
  
plt.subplot(212)  
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')  
plt.show()
```

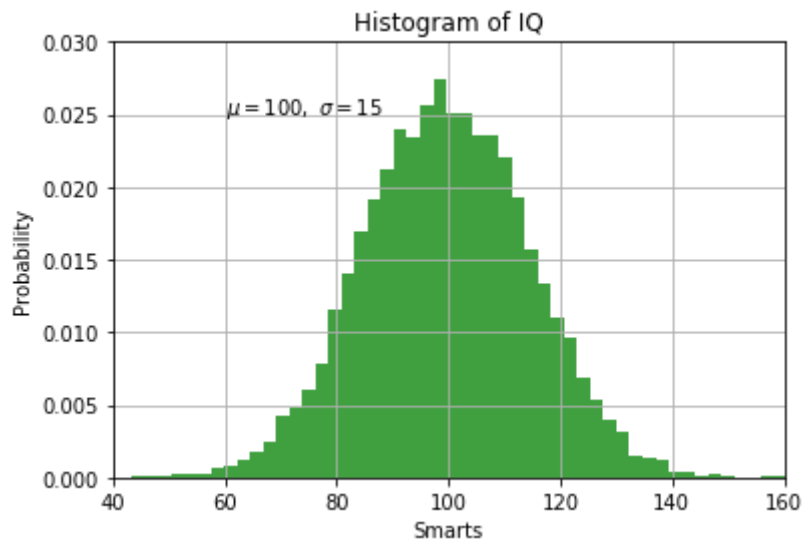



```
In [175]: np.random.seed(19680801)

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



```

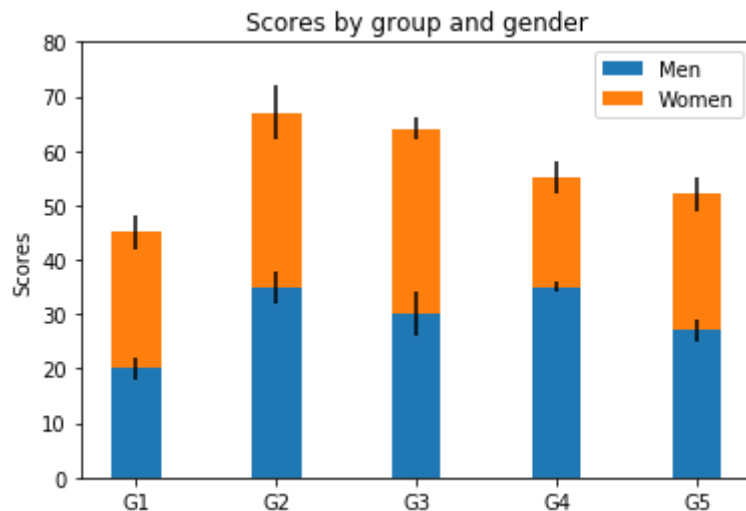
In [176]: N = 5
menMeans = (20, 35, 30, 35, 27)
womenMeans = (25, 32, 34, 20, 25)
menStd = (2, 3, 4, 1, 2)
womenStd = (3, 5, 2, 3, 3)
ind = np.arange(N)      # the x locations for the groups
width = 0.35            # the width of the bars: can also be len(x) sequence

p1 = plt.bar(ind, menMeans, width, yerr=menStd)
p2 = plt.bar(ind, womenMeans, width,
              bottom=menMeans, yerr=womenStd)

plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(ind, ('G1', 'G2', 'G3', 'G4', 'G5'))
plt.yticks(np.arange(0, 81, 10))
plt.legend((p1[0], p2[0]), ('Men', 'Women'))

plt.show()

```



```
In [177]: labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
          sizes = [15, 30, 45, 10]
          explode = (0, 0.1, 0, 0) # only "explode" the 2nd slice (i.e. 'Hogs')

          fig1, ax1 = plt.subplots()
          ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
                  shadow=True, startangle=90)
          ax1.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.

          plt.show()
```

