Chapter 1: Getting Started with Python Language

Python 3.x

Version	*Release Date*
3.8	2020-04-29
3.7	2018-06-27
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Version	*Release Date*
2.7	2010-07-03
2.6	2008-10-02
2.5	2006-09-19
2.4	2004-11-30
2.3	2003-07-29
2.2	2001-12-21
2.1	2001-04-15
2.0	2000-10-16

Section 1.1: Getting Started

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Two major versions of Python are currently in active use:

```
* Python 3.x is the current version and is under active development.
* Python 2.x is the legacy version and will receive only security updates until 202
0. No new features will be implemented. Note that many projects still use Python 2,
although migrating to Python 3 is getting easier.
```

You can download and install either version of Python here. See Python 3 vs. Python 2 for a comparison between them. In addition, some third-parties offer re-packaged versions of Python that add commonly used libraries and other features to ease setup for common use cases, such as math, data analysis or scientific use. See the list at the official site.

Verify if Python is installed

To confirm that Python was installed correctly, you can verify that by running the following command in your favorite terminal (If you are using Windows OS, you need to add path of python to the environment variable before using it in command prompt):

```
 $ python --version
Python 3.x
```

Hello, World in Python using IDLE

IDLE is a simple editor for Python, that comes bundled with Python.

How to create Hello, World program in IDLE

```
* Open IDLE on your system of choice.
    - In older versions of Windows, it can be found at All Programs under the Windo
ws menu.
    - In Windows 8+, search for IDLE or find it in the apps that are present in you
r system.
    - On Unix-based (including Mac) systems you can open it from the shell by typin
g $ idle python_file.py
* It will open a shell with options along the top.<br>
```

In the shell, there is a prompt of three right angle brackets:

```
>>>
```

Now write the following code in the prompt:

```
>>>print("Hello World")
```

Note: There are multiple ways to create and run a file, I'm using Jupyter Notebook

Section 1.2: Creating variables and assigning values

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

=

In [1]:
```
# Integer
a = 2
print(a)
```

```
2
```

In [2]:
```
# Integer
b = 456789045678945678
print(b)
```

```
456789045678945678
```

In [3]:
```
# Floating point
pi = 3.14
print(pi)
```

```
3.14
```

In [4]:
```
# String
c = 'A'
print(c)
```

```
A
```

In [5]:
```
# String
name = 'Sinku Kumar'
print(name)
```

```
Sinku Kumar
```

In [6]:
```
# Empty value or null data type
x = None
print(x)
```

```
None
```

In [7]:
```
# Wring assgnment of Variable
0 = x
```

```
  File "<ipython-input-7-4c76e9e4e4b7>", line 2
    0 = x
         ^
SyntaxError: can't assign to literal
```

You can not use python's keywords as a valid variable name. Yo can see the list of keywords by:

In [8]:
```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'fo
r', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'no
t', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Rules for variable naming:

```
1. Variables names must start with a letter or an underscore.
```

In [9]:
```
x = True # valid
_y = True # valid

9x = False # starts with numeral
```

```
  File "<ipython-input-9-43b5f744b991>", line 4
    9x = False # starts with numeral
     ^
SyntaxError: invalid syntax
```

In [10]:
```
$y = False # startes with symbol
```

```
  File "<ipython-input-10-6540f6603f03>", line 1
    $y = False # startes with symbol
    ^
SyntaxError: invalid syntax
```

```
2. The remainder of your variable name may consist of letters, numbers and undersco
res.
```

In [11]:
```
has_0_in_it = True # still valid
```

```
3. The names are case sensitive
```

In [12]:
```
x = 9
X = 10
print(x)
print(X)
```

```
9
10
```

Even though there's no need to specify a data type when declaring a variable in Python, while allocating the necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:

In [13]:
```
a = 2
print(type(a))
```

```
<class 'int'>
```

In [14]:
```
b = 3456789043567890
print(type(b))
```

```
<class 'int'>
```

```
In [15]: pi = 3.14
         print(type(pi))

         <class 'float'>
```

```
In [16]: c = 'A'
         print(type(c))

         <class 'str'>
```

```
In [17]: name = "Sinku Kumar"
         print(type(name))

         <class 'str'>
```

```
In [18]: q = True
         print(type(q))

         <class 'bool'>
```

```
In [19]: x = None
         print(type(x))

         <class 'NoneType'>
```

You can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the = operator:

```
In [20]: a, b, c = 1,2,3
         print(a, b, c)

         1 2 3
```

```
In [21]: a, b, c = 1,2

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         <ipython-input-21-43420f4e6418> in <module>
         ----> 1 a, b, c = 1,2

         ValueError: not enough values to unpack (expected 3, got 2)
```

```
In [22]: a, b = 1, 2, 3

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         <ipython-input-22-8904fd2ea925> in <module>
         ----> 1 a, b = 1, 2, 3

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

```
In [23]: a, b, _ = 1,2,3
         print(a,b,_)
```

```
1 2 3
```

```
In [24]: a = b = c = 1
         print(a,b,c)
```

```
1 1 1
```

```
In [25]: b = 2
         print(b)
```

```
2
```

The above is also true for mutable types (like list, dict, etc.) just as it is true for immutable types (like int, string, tuple, etc.):

```
In [26]: x = y = [7,8,9]
         x = [13,8,9]
         print(x)
         print(y)
```

```
[13, 8, 9]
[7, 8, 9]
```

```
In [27]: x = y = [5,6,7]
         x[0] = 13
         print(x)
         print(y)
```

```
[13, 6, 7]
[13, 6, 7]
```

Section 1.3 Block Indentation

Python uses indentation to define control and loop constructs. This contributes to Python's readability, however, it requires the programmer to pay close attention to the use of whitespace. Thus, editor miscalibration could result in code that behaves in unexpected ways.

Python uses the colon symbol (:) and indentation for showing where blocks of code begin and end (If you come from another language, do not confuse this with somehow being related to the ternary operator). That is, blocks in Python, such as functions, loops, if clauses and other constructs, have no ending identifiers. All blocks start with a colon and then contain the indented lines below it.

For example:

```
In [28]: def my_function():
             a = 2
             return(a)
         print(my_function())
```

```
2
```

or

```
In [29]: if a >b:
             print(a)
         else:
             print(b)
```

```
2
```

Blocks that contain exactly one single-line statement may be put on the same line, though this form is generally not considered good style:

```
In [30]: if a > b:     print(a)
         else: print(b)
```

```
2
```

Attempting to do this with more than a single statement will not work:

```
In [31]: if x >y:     y = x
             print(y)
```

```
  File "<ipython-input-31-2b6def82f9ce>", line 2
    print(y)
    ^
IndentationError: unexpected indent
```

```
In [32]: if x > y: while y != z: y-=1
```

```
  File "<ipython-input-32-ef042b86af11>", line 1
    if x > y: while y != z: y-=1
                  ^
SyntaxError: invalid syntax
```

An empty block causes an IndentationError. Use pass (a command that does nothing) when you have a block with no content:

```
In [33]: def will_be_implemented_later():
             pass
```

Spaces vs Tabs

In short: always use 4 spaces for indentation.
Using tabs exclusively is possible but PEP 8, the style guide for Python code, states that spaces are preferred.
Python 3.x Version ≥ 3.0
Python 3 disallows mixing the use of tabs and spaces for indentation. In such case a compile-time error is generated: Inconsistent use of tabs and spaces in indentation and the program will not run.

Many editors have "tabs to spaces" configuration. When configuring the editor, one should differentiate between the tab character ('\t') and the Tab key.

```
* The tab character should be configured to show 8 spaces, to match the language se
mantics - at least in cases when (accidental) mixed indentation is possible. Editor
s can also automatically convert the tab character to spaces.
* However, it might be helpful to configure the editor so that pressing the Tab key
will insert 4 spaces, instead of inserting a tab character.
```

Python source code written with a mix of tabs and spaces, or with non-standard number of indentation spaces can be made pep8-conformant using autopep8. (A less powerful alternative comes with most Python installations: reindent.py)

Section 1.4: Datatypes

Built-in Types

Booleans

bool: A boolean value of either True or False. Logical ooperations like and, or, not can be performed on booleans.

In [34]:
```
x or y  # if x is False then y, otherwise x
x and y # if x is False then x, otherwise y
not x   # if x is True then False, otherwise True
```

Out[34]:
```
False
```

In Python 2.x and in Python 3.x, a boolean is also an int. The bool type is a subclass of the int type and True and False are its only instances:

```
In [35]: issubclass(bool, int)  # True
         isinstance(True, bool) # True
         isinstance(False, bool)# True
```

```
Out[35]: True
```

```
In [36]: issubclass(float, int)
```

```
Out[36]: False
```

If boolean values are used in arithmetic operations, their integer values (1 and 0 for True and False) will be used to return an integer result:

```
In [37]: True + False == 1 # 1 + 0 == 1
         True * True == 1  # 1 * 1 == 1
```

```
Out[37]: True
```

Numbers

- int: Integer number

```
In [38]: a = 2
         b = 100
         c = 123456789
         d = 38435678902432423432
```

Integers in Python are of arbitrary sizes.

Note: in older versions of Python, a long type was available and this was distinct from int. The two have been unified.

- float: Floating point number; precision depends on the implementation and system architecture, for CPython the float datatype corresponds to a C double.

```
In [39]: a = 2.0
         b = 100.e9
         c = 123456789.e3
         print(a, b, c)
```

```
2.0 100000000000.0 123456789000.0
```

- complex: Complex numbers

```
In [40]: a = 2 + 1j
         b = 100 + 10j
```

The <, <=, and >= operators will raise a TypeError exception when any operand is complex number.

Strings

Python 3.x

- str: a unicode string. The type of 'hello'
- bytes: a byte string: The type of b 'hello'

Sequences and collections

Python differentiates between oerdered sequnces and unordered collections (such as set and dict).

- strings(str, bytes, unicode) are sequnces
- reversed: A reversed order of str with reversed function

```
In [41]: a = reversed('hello')
         print(a)
```

```
<reversed object at 0x000001EFE78358C8>
```

- tuple: An ordered collection of n values of any type(n >= 0).

```
In [42]: a = (1, 2, 3)
         b = ('a', 1, 'python', (1,2))
         b[2] = 'something else' # returns a TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-42-8ec7f114d04f> in <module>
      1 a = (1, 2, 3)
      2 b = ('a', 1, 'python', (1,2))
----> 3 b[2] = 'something else' # returns a TypeError

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

Supports indexing; immutable; hashable if all its members are hashable

- list: An unordered collection of n values(n >= 0)

```
In [43]: a = [1, 2, 3]
         b = ['a', 1, 'python', (1,2), [1,2]]
         b[2] = 'something else' # allowed
         print(b)
```

```
['a', 1, 'something else', (1, 2), [1, 2]]
```

Not hashtable; mutable.

- set: an unordered collection of unique values. Items must be hashtable.

```
In [44]: a = {1, 2, 'a'}
         print(a)
```

```
{1, 2, 'a'}
```

- dict: An unordered collection of unique key-value pairs; keys must be hashtable

```
In [45]: a = {1: 'one',
              2: 'two'}
         b = {'a': [1,2,3],
              'b': 'a string'}
         print(a)
         print(b)
         print(a[1])
         print(b['a'])
```

```
{1: 'one', 2: 'two'}
{'a': [1, 2, 3], 'b': 'a string'}
one
[1, 2, 3]
```

An object is hashable if it has a hash value which never changes during its lifetime (it needs a **hash**() method), and can be compared to other objects (it needs an **eq**() method). Hashable objects which compare equality must have the same hash value.

Built-in constants

In conjunction with the built-in datatypes there are a small number of built-in constants in the built-in namespace:

- True: The true value of the built-in type bool
- False: The false value of the built-in type bool
- None: A singleton object used to signal that a value is absent.
- Ellipsis or ...: used in core Python3+ anywhere and limited usage in Python2.7+ as part of array notation. numpy and related packages use this as a 'include everything' reference in arrays.
- NotImplemented: a singleton used to indicate to Python that a special method doesn't support the specific arguments, and Python will try alternatives if available.

In [46]:
```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x

None doesn't have any natural ordering. Using ordering comparison operators (<, <=, >=, >) isn't supported anymore and will raise a TypeError.

Testing the type of variables

In python, we can check the datatype of an object using the built-in function type.

In [47]:
```
a = '123'
b = 123
print(type(a))
print(type(b))
```

```
<class 'str'>
<class 'int'>
```

In conditional statements it is possible to test the datatype with isinstance. However, it is usually not encouraged to rely on the type of the variable.

In [48]:
```
i = 7
if isinstance(i, int):
    i += int(i)
elif isinstance(i, str):
    i = int(i)
    i += 1
```

For information on the differences between type() and isinstance() read: Differences between isinstance and type in Python on stackoverflow.

To test if something is of NoneType:

```
In [49]: x = None
         if x is None:
             print('Not a surprise, I just defined x as None.')
```

```
Not a surprise, I just defined x as None.
```

Converting between datatypes

You can perform explicit datatype conversion.

For example, '123 is of str type and it can be converted to integer' using int function.

```
In [50]: a = '123'
         b = int(a)
         print(type(a))
         print(type(b))
```

```
<class 'str'>
<class 'int'>
```

Converting from a float string such as '123.456' canbe done using float function.

```
In [51]: a = 123.456
         b = float(a)
         c = int(a)
         d = int(b)
         print(a,b,c,d)
```

```
123.456 123.456 123 123
```

You can also convert sequences or collection types

```
In [52]: a = 'hello'
         print(list(a))
         print(set(a))
         print(tuple(a))
```

```
['h', 'e', 'l', 'l', 'o']
{'h', 'l', 'e', 'o'}
('h', 'e', 'l', 'l', 'o')
```

Explicit string type at definition of literals

With one letter labels just in front of the quotes you can tell what type of string you want to define.

- b'foo bar': results bytes in Python2, str in Python 2
- u'foo bar': results str in Python 3, unicode in Python 2
- 'foo bar': results str
- r'foo bar': results so called raw string, where escaping special characters is not necessary, everything is taken verbatim as you typed

In [53]:
```
normal = 'foo\nbar'
print(normal)
escaped = 'foo\\nbar'
print(escaped)
raw = r'foo\\nbar'
print(raw)
```

```
foo
bar
foo\nbar
foo\\nbar
```

Mutable and Immutable Data Types

An object is called mutable if it can be changed. For example, when you pass a list to some function, the list can be changed:

In [54]:
```
def f(m):
    m.append(3) # adds a number to the list. This is a mutation.

x = [1,2]
f(x)
x == [1,2]      # False now, since an item was added to the list
```

Out[54]:
```
False
```

An object is called immutable if it cannot be changed in any way. For example, integers are immutable, since there's no way to change them:

In [55]:
```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2) # Will always be True, since no function can change the object
(1, 2)
```

Note that variables themselves are mutable, so we can reassign the variable x, but this does not change the object that x had previously pointed to. It only made x point to a new object.

Data types whose instances are mutable are called mutable data types, and similarly for immutable objects and datatypes.

Examples of immutable Data Types:

```
* int, long, float, complex
* str
* bytes
* tuple
* frozenset
```

Examples of mutable Data Types:

```
* bytearray
* list
* set
* dict
```

Section 1.5: Collection Types

There are a number of collection types in Python. While types such as int str hold a single value, collection types hold multiple values.

Lists

The list type is probably the most commonly used collection type in Python. Despite its name, a list is more like an array in other languages, mostly JavaScript, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, seperated by commas, in square brackets:

In [56]:
```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

A list can be empty

In [57]:
```
empty_list = []
```

The elements of list are not restricted to a single data type, which makes sense given that Python is adynamic language:

```
In [58]: nested_list = [[a, b, c], [1, 2, 3]]
```

The elements of a list can be accesses via index, or numeric representation of their position. Lists in Python are zero-indexed meaning that the first element in the list at index 0, the second is at index 1 and so on:

```
In [59]: names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
         print(names[0])
         print(names[1])
```

```
Alice
Bob
```

Indices can also be negative which means counting from the end of the list (-1 being the index of the last element). So, using the list from the above example:

```
In [60]: print(names[-1])
         print(names[-4])
```

```
Eric
Bob
```

Lists are mutable, so you can change the values in a list:

```
In [61]: names[0] = 'Sachin'
         print(names)
```

```
['Sachin', 'Bob', 'Craig', 'Diana', 'Eric']
```

Besides, it is possible to add and/or remove elements from a list:

Append object to end of list with L.append(object), returns None.

```
In [62]: names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
         names.append("Sia")
         print(names)
```

```
['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Add a new element to list at a specific index. L.insert(index, object)

```
In [63]: names.insert(1, "Nikki")
         print(names)

['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Remove the first occurrence of a value with L.remove(value), returns None

```
In [64]: names.remove("Bob")
         print(names)

['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Get the index in the list of the first item whose value is x. It will show an error if there is no such item

```
In [65]: names.index("Alice")

Out[65]: 0
```

Count length of list

```
In [66]: len(names)

Out[66]: 6
```

Count occurrence of any item in list

```
In [67]: a = [1, 1, 1, 2, 3, 4]
         a.count(1)

Out[67]: 3
```

Reverse the list

```
In [68]: a = [1, 1, 1, 2, 3, 4]
         reversed_a = a.reverse()
         print(reversed_a)

None
```

```
In [69]: a = [1, 1, 1, 2, 3, 4]
         a[::-1]

Out[69]: [4, 3, 2, 1, 1, 1]
```

Remove and return item at index (defaults to the last item) with L.pop([index]), returns the item

In [70]:
```
names.pop()
```
Out[70]:
```
'Sia'
```

You can iterate over the list elements like below:

In [71]:
```
my_list = ['a', 'b', 'c', 1, 2, 3]
for element in my_list:
    print(element)
```
```
a
b
c
1
2
3
```

Tuples

A tuple is similar to a list except that it is fixed-length and immutable. So the values in the tuple cannot be changed nor the values be added to or removed from the tuple. Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port. Tuples are represented with parentheses instead of square brackets:

In [72]:
```
ip_address = ('10.20.30.40', 8080)
```

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid

In [73]:
```
one_member_tuple = ('Only member', )
# or
one_member_tuple = 'Only member', # No brackets
# or just using tuple syntax
one_member_tuple = tuple(['Only Member'])
```

Dictionaries

A dictionary in Python is a collection of key-value pairs. The dictionary is surrounded by curly braces. Each pair is separated by a comma and the key and value are separated by a colon. Here is an example:

In [74]:
```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

To get a value, refer to it by its key:

In [75]:
```
ca_capital = state_capitals['California']
print(ca_capital)
```

```
Sacramento
```

You can also get all of the keys in a dictionary and then iterate over them:

In [76]:
```
for k in state_capitals.keys():
 print('{} is the capital of {}'.format(state_capitals[k], k))
```

```
Little Rock is the capital of Arkansas
Denver is the capital of Colorado
Sacramento is the capital of California
Atlanta is the capital of Georgia
```

Dictionaries strongly resemble JSON syntax. The native json module in the Python standard library can be used to convert between JSON and dictionaries.

set

A set is a collection of elements with no repeats and without insertion order but sorted order. They are used in situations where it is only important that some things are grouped together, and not what order they were included. For large groups of data, it is much faster to check whether or not an element is in a set than it is to do the same for a list.

Defining a set is very similar to defining a dictionary:

In [77]:
```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Or you can build a set using an existing list:

In [78]:
```
my_list = [1,2,3]
my_set = set(my_list)
```

Check membership of the set using in:

In [79]:
```
if name in first_names:
    print(name)
```

You can iterate over a set exactly like a list, but remember: the values will be in an arbitrary, implementationdefined order.

defaultdict

A defaultdict is a dictionary with a default value for keys, so that keys for which no value has been explicitly defined can be accessed without errors. defaultdict is especially useful when the values in the dictionary are collections (lists, dicts, etc) in the sense that it does not need to be initialized every time when a new key is used.

A defaultdict will never raise a KeyError. Any key that does not exist gets the default value returned.

For example, consider the following dictionary

```
In [80]: state_capitals = {
             'Arkansas': 'Little Rock',
             'Colorado': 'Denver',
             'California': 'Sacramento',
             'Georgia': 'Atlanta'
         }
```

If we try to access a non-existent key, python returns us an error as follows

```
In [81]: state_capitals['Alabama']

         ---------------------------------------------------------------------------
         KeyError                                  Traceback (most recent call last)
         <ipython-input-81-9482349d4814> in <module>
         ----> 1 state_capitals['Alabama']

         KeyError: 'Alabama'
```

Let us try with a defaultdict. It can be found in the collections module.

```
In [82]: from collections import defaultdict
         state_capitals = defaultdict(lambda: 'Boston')
         state_capitals['Alabama']

Out[82]: 'Boston'
```

What we did here is to set a default value(Boston) in case the given key does not exist. Now populate the dict as before:

```
In [83]: state_capitals['Arkansas'] = 'Little Rock'
         state_capitals['California'] = 'Sacramento'
         state_capitals['Colorado'] = 'Denver'
         state_capitals['Georgia'] = 'Atlanta'
```

If we try to access the dict with a non-existent key, python will return us the default value i.e. Boston

```
In [84]: state_capitals['Alabama']
Out[84]: 'Boston'
```

and returns the created values for existing key just like a normal dictionary

```
In [85]: state_capitals['Arkansas']
Out[85]: 'Little Rock'
```

Section 1.6: IDLE - Python GUI

IDLE is Python's Integrated Development and Learning Environment and is an alternative to the command line. As the name may imply, IDLE is very useful for developing new code or learning python. On Windows this comes with the Python interpreter, but in other operating systems you may need to install it through your package manager.

The main purposes of IDLE are:

- Multi-window text editor with syntax highlighting, autocompletion, and smart indent
- Python shell with syntax highlighting
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility
- Automatic indentation (useful for beginners learning about Python's indentation)
- Saving the Python program as .py files and run them and edit them later at any them using IDLE.

In IDLE, hit F5 or run Python Shell to launch an interpreter. Using IDLE can be a better learning experience for new users because code is interpreted as the user writes.

Section 1.7: User Input

Interactive input

To get input from the user, use the input function (note: in Python 2.x, the function is called raw_input instead, although Python 2.x has its own version of input that is completely different):

Python 3.x

```
In [86]: name = input("What is your name? ")
         print("Hello", name)

         What is your name?
         Hello
```

Note that the input is always of type str, which is important if you want the user to enter numbers. Therefore, you need to convert the str before trying to use it as a number:

```
In [87]: x = int(input("Write a number: "))
         x / 2
         float(x) / 2
```

```
Write a number:

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-87-6080654e2b3c> in <module>
----> 1 x = int(input("Write a number: "))
      2 x / 2
      3 float(x) / 2

ValueError: invalid literal for int() with base 10: ''
```

NB: It's recommended to use try/except blocks to catch exceptions when dealing with user inputs. For instance, if your code wants to cast a raw_input into an int, and what the user writes is uncastable, it raises a ValueError.

Section 1.8: Built in Modules and Functions

A module is a file containing Python definitions and statements. Function is a peice of code which execute some logic.

```
In [88]: pow(2,3)
Out[88]: 8
```

To check the built in function in python we can use dir(). If called without an argument, return the names in the current scope. Else, return an alphabetized list of names comprising (some of) the attribute of the given object, and of attributes reachable from it.

```
In [89]: dir(__builtins__)
```

```
Out[89]: ['ArithmeticError',
          'AssertionError',
          'AttributeError',
          'BaseException',
          'BlockingIOError',
          'BrokenPipeError',
          'BufferError',
          'BytesWarning',
          'ChildProcessError',
          'ConnectionAbortedError',
          'ConnectionError',
          'ConnectionRefusedError',
          'ConnectionResetError',
          'DeprecationWarning',
          'EOFError',
          'Ellipsis',
          'EnvironmentError',
          'Exception',
          'False',
          'FileExistsError',
          'FileNotFoundError',
          'FloatingPointError',
          'FutureWarning',
          'GeneratorExit',
          'IOError',
          'ImportError',
          'ImportWarning',
          'IndentationError',
          'IndexError',
          'InterruptedError',
          'IsADirectoryError',
          'KeyError',
          'KeyboardInterrupt',
          'LookupError',
          'MemoryError',
          'ModuleNotFoundError',
          'NameError',
          'None',
          'NotADirectoryError',
          'NotImplemented',
          'NotImplementedError',
          'OSError',
          'OverflowError',
          'PendingDeprecationWarning',
          'PermissionError',
          'ProcessLookupError',
          'RecursionError',
          'ReferenceError',
          'ResourceWarning',
          'RuntimeError',
          'RuntimeWarning',
          'StopAsyncIteration',
          'StopIteration',
          'SyntaxError',
          'SyntaxWarning',
          'SystemError',
          'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'WindowsError',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'vars',
'zip']
```

To know the functionality of any function, we can use built in function help .

```
In [90]: help(max)
```

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Built in modules contains extra functionalities. For example to get a square root of a number we need to include math module.

```
In [91]: import math
         math.sqrt(16)
```

```
Out[91]: 4.0
```

To know all the functions in a module we can assign the functions list to a variable, and then print the variable

```
In [92]: import math
         dir(math)
```

```
Out[92]: ['__doc__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          'acos',
          'acosh',
          'asin',
          'asinh',
          'atan',
          'atan2',
          'atanh',
          'ceil',
          'copysign',
          'cos',
          'cosh',
          'degrees',
          'e',
          'erf',
          'erfc',
          'exp',
          'expm1',
          'fabs',
          'factorial',
          'floor',
          'fmod',
          'frexp',
          'fsum',
          'gamma',
          'gcd',
          'hypot',
          'inf',
          'isclose',
          'isfinite',
          'isinf',
          'isnan',
          'ldexp',
          'lgamma',
          'log',
          'log10',
          'log1p',
          'log2',
          'modf',
          'nan',
          'pi',
          'pow',
          'radians',
          'remainder',
          'sin',
          'sinh',
          'sqrt',
          'tan',
          'tanh',
          'tau',
          'trunc']
```

it seems **doc** is useful to provide some documentation in, say, functions

In [93]:
```
math.__doc__
```

Out[93]:
```
'This module provides access to the mathematical functions\ndefined by the C
standard.'
```

In addition to functions, documentation can also be provided in modules. So, if you have a file named helloWorld.py like this:

In [94]:
```
"""This is the module docstring."""

def sayHello():
     """This is the function docstring."""
     return 'Hello World'
```

You can access its docstrings like this:

In [95]:
```
import helloWorld
helloWorld.__doc__
```

Out[95]:
```
'This is the module docstring.'
```

In [96]:
```
helloWorld.sayHello.__doc__
```

Out[96]:
```
'This is the function docstring.'
```

For any user defined type, its attributes, its class's attributes, and recursively the attributes of its class's base classes can be retrieved using dir()

In [97]:
```
class MyClassObject(object):
    "This is an empty function."
    pass
```

```
In [98]: dir(MyClassObject)
```

```
Out[98]: ['__class__',
          '__delattr__',
          '__dict__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__le__',
          '__lt__',
          '__module__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__setattr__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          '__weakref__']
```

```
In [99]: MyClassObject.__doc__
```

```
Out[99]: 'This is an empty function.'
```

Any data type can be simply converted to string using a builtin function called str. This function is called by default when a data type is passed to print

```
In [100]: str(123)
```

```
Out[100]: '123'
```

Section 1.9: Creating a module

A Module is an importable file containing definitions and statements.

A module can be created by creating a .py file

Functions in a module can be used by importing the module.

For modules that you have made, they will need to be in the same directory as per the file that you are importing them into. (However, you can put them into the Python lib directory with the pre-included modules, but should be avoided if possible.)

In [101]:
```
import hello
hello.say_hello()
```
```
Hello!
```

Modules can be imported by other modules.

In [102]:
```
# greet.py
import hello
hello.say_hello()
```
```
Hello!
```

Specific functions of a module can be imported.

In [103]:
```
# greet.py
from hello import say_hello
say_hello()
```
```
Hello!
```

Modules can be aliased.

In [104]:
```
# greet.py
import hello as ai
ai.say_hello()
```
```
Hello!
```

A module can be stand-alone runnable script.

In [105]:
```
# run hello.py
if __name__ == '__main__':
    from hello import say_hello
    say_hello()
```
```
Hello!
```

If the module is inside a directory and needs to be detected by python, the directory should contain a file named **init**.py.

Section 1.10: Installation of Python

Will be added later

Section 1.11: String function - str() and repr()

There are two functions that can be used to obtain a readble representation of an object.

- repr(x) calls x.**repr**(): a representation of x. eval will usually convert the result of this function back to the original object.
- str(x) calls x.**str**(): a human-readable string that describes the object. This may elide some technical detail.

repr()

For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(). Otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object along with additional information. This often includes the name and address of the object.

str()

For strings, this returns the string itself. The difference between this and repr(object) is that str(object) does not always attempt to return a string that is acceptable to eval(). Rather, its goal is to return a printable or 'human readable' string. If no argument is given, this returns the empty string, ''.

Example 1:

```
In [106]: s = """w'o"w"""
          repr(s)

Out[106]: '\'w\\\'o"w\''
```

```
In [107]: joo = "JooJoo"
          j = joo
          repr(j)

Out[107]: "'JooJoo'"
```

```
In [108]: str(j)
          eval(repr(j)) == 'Lol' # True when 'JooJoo'

Out[108]: False
```

Example 2

```
In [109]: import datetime
          today = datetime.datetime.now()
          str(today)

Out[109]: '2020-08-23 13:10:00.315291'
```

In [110]:
```
repr(today)
```
Out[110]:
```
'datetime.datetime(2020, 8, 23, 13, 10, 0, 315291)'
```

When writing a class, you can override these methods to do whatever you want:

In [111]:
```
class Represent(object):

    def __init__(self, x,y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y=\"{}\")".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

Using the above class we can see the results

In [112]:
```
r = Represent(1, "Hopper")
print(r)
print(r.__repr__)
rep = r.__repr__()
print(rep)
r2 = eval(rep)
print(r2)
print(r2 == r)
```
```
Representing x as 1 and y as Hopper
<bound method Represent.__repr__ of Represent(x=1,y="Hopper")>
Represent(x=1,y="Hopper")
Representing x as 1 and y as Hopper
False
```

Section 1.12: Installing external modules using pip

pip is your friend when you need to install any package from plethora of choices availiable at the python package index(PyPi). pip is already installed if you're using Python 3 >=3.4 downloaded from python.org.

```
$ pip search [query]
$ pip install [package_name]
$ pip install [package_name]==x.x.x
$ pip install [package_name]>=x.x.x
```

where x.x.x is the version number of the package you want to install.

Upgrading installed packages

When new versions of installed packages appear they are not automatically installed to your system. To get an overview of which of your installed packages have become outdated, run:

```
$ pip list --outdated
```

To upgrade a specific package use

```
$ pip install [package_name] --upgrade
```

Updating all outdates packages is not a standard functionality of pip.

Upgrading pip

You can upgrade your existing pip installation by using the following commands

```
* On Linux or macOS X:
    <code>$ pip install -U pip</code>

* On Windows:

    $ py -m pip install -U pip

    or

    $ python -m pip install -U pip
```

Section 1.13: Help Utility

Python has several function built into the interpreter. If you want to get information of keywords, built-in functions, modules or topics open Python console and enter:

In [113]:
```
help()
```

```
Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

You will receive information by entering keywords directly:

In [114]:
```
# help(help) # not executing
```

or within the utility

In [115]:
```
# help> help # not executing
```

In [116]:

```
import math
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.

        This is the smallest integer >= x.

    copysign(x, y, /)
        Return a float with the magnitude (absolute value) of x but the sign
of y.

        On platforms that support signed zeros, copysign(1.0, -0.0)
        returns -1.0.

    cos(x, /)
        Return the cosine of x (measured in radians).

    cosh(x, /)
        Return the hyperbolic cosine of x.

    degrees(x, /)
        Convert angle x from radians to degrees.

    erf(x, /)
        Error function at x.
    erfc(x, /)
        Complementary error function at x.

    exp(x, /)
        Return e raised to the power of x.

    expm1(x, /)
        Return exp(x)-1.

        This function avoids the loss of precision involved in the direct eva
luation of exp(x)-1 for small x.

    fabs(x, /)
        Return the absolute value of the float x.

    factorial(x, /)
        Find x!.

        Raise a ValueError if x is negative or non-integral.

    floor(x, /)
        Return the floor of x as an Integral.

        This is the largest integer <= x.

    fmod(x, y, /)
        Return fmod(x, y), according to platform C.

        x % y may differ.

    frexp(x, /)
        Return the mantissa and exponent of x, as pair (m, e).

        m is a float and e is an int, such that x = m * 2.**e.
        If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.

    fsum(seq, /)
        Return an accurate floating point sum of values in the iterable seq.

        Assumes IEEE-754 floating point arithmetic.

    gamma(x, /)
        Gamma function at x.

    gcd(x, y, /)
        greatest common divisor of x and y

    hypot(x, y, /)
        Return the Euclidean distance, sqrt(x*x + y*y).

    isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
        Determine whether two floating point numbers are close in value.

          rel_tol
            maximum difference for being considered "close", relative to the
            magnitude of the input values
          abs_tol
```

```
                maximum difference for being considered "close", regardless of th
e
                magnitude of the input values

            Return True if a is close in value to b, and False otherwise.

            For the values to be considered close, the difference between them
            must be smaller than at least one of the tolerances.

            -inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
            is, NaN is not close to anything, even itself.  inf and -inf are
            only close to themselves.

        isfinite(x, /)
            Return True if x is neither an infinity nor a NaN, and False otherwis
e.

        isinf(x, /)
            Return True if x is a positive or negative infinity, and False otherw
ise.

        isnan(x, /)
            Return True if x is a NaN (not a number), and False otherwise.

        ldexp(x, i, /)
            Return x * (2**i).

            This is essentially the inverse of frexp().

        lgamma(x, /)
            Natural logarithm of absolute value of Gamma function at x.

        log(...)
            log(x, [base=math.e])
            Return the logarithm of x to the given base.

            If the base not specified, returns the natural logarithm (base e) of
x.

        log10(x, /)
            Return the base 10 logarithm of x.

        log1p(x, /)
            Return the natural logarithm of 1+x (base e).

            The result is computed in a way which is accurate for x near zero.

        log2(x, /)
            Return the base 2 logarithm of x.

        modf(x, /)
            Return the fractional and integer parts of x.

            Both results carry the sign of x and are floats.

        pow(x, y, /)
            Return x**y (x to the power of y).
        radians(x, /)
            Convert angle x from degrees to radians.

        remainder(x, y, /)
            Difference between x and the closest integer multiple of y.

            Return x - n*y where n*y is the closest integer multiple of y.
            In the case where x is exactly halfway between two multiples of
            y, the nearest even value of n is used. The result is always exact.

        sin(x, /)
            Return the sine of x (measured in radians).

        sinh(x, /)
            Return the hyperbolic sine of x.

        sqrt(x, /)
            Return the square root of x.

        tan(x, /)
            Return the tangent of x (measured in radians).

        tanh(x, /)
            Return the hyperbolic tangent of x.

        trunc(x, /)
            Truncates the Real x to the nearest Integral toward 0.

            Uses the __trunc__ magic method.

    DATA
        e = 2.718281828459045
        inf = inf
        nan = nan
        pi = 3.141592653589793
        tau = 6.283185307179586

    FILE
        (built-in)
```

An now you will get a list of the available methods in the module, bu only AFTER you have imported it.

Close the helper with quit.

```
In [ ]:
```