Getting started with Python Language Introduction Topic

Examples

Getting Started

Python is a high-level, structured, open-source, dynamically typed programming language.

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 2020. 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:

```
$ python --version
```

Python 3.x ≥3.0

If you have *Python 3* installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
Python 3.6.0
```

Python 2.x ≤2.7

If you have *Python 2* installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
Python 2.7.13
```

If you have installed Python 3, but `$ python --version` outputs a Python 2 version, you also have Python 2 installed. This is often the case on MacOS, and many Linux distributions. Use `$ python3` instead to explicitly use the Python 3 interpreter.

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 Windows menu.
  - In Windows 8+, search for `IDLE` or find it in the apps that are present in your system.
  - On Unix-based (including Mac) systems you can open it from the shell by typing `$ idle python_file.py` .
- It will open a shell with options along the top.

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

```
>>>
```

Now write the following code in the prompt:

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

Hit Enter .

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

Hello World Python file

Create a new file `hello.py` that contains the following line:

Python 3.x ≥3.0

```
print('Hello, World')
```

Python 2.x ≥2.6

You can use the Python 3 print function in Python 2 with the following import statement:

```
from __future__ import print_function
```

Python 2 has a number of functionalities that can be optionally imported from Python 3 using the __future__ module, as discussed here .

Python 2.x ≤2.7

If using Python 2, you may also type the line below. Note that this is not valid in Python 3 and thus not recommended because it reduces cross-version code compatibility.

```
print 'Hello, World'
```

In your terminal, navigate to the directory containing the file hello.py .

Type python hello.py , then hit the Enter key.

```
$ python hello.py
Hello, World
```

You should see Hello, World printed to the console.

You can also substitute hello.py with the path to your file. For example, if you have the file in your home directory and your user is "user" on Linux, you can type python /home/user/hello.py .

Launch an interactive Python shell

By executing (running) the python command in your terminal, you are presented with an interactive Python shell. This is also known as the Python Interpreter or a REPL (for 'Read Evaluate Print Loop').

```
$ python
Python 2.7.12 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, World'
Hello, World
>>>
```

If you want to run Python 3 from your terminal, execute the command python3 .

```
$ python3
Python 3.5.2 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World')
Hello, World
>>>
```

Alternatively, start the interactive prompt and load file with python -i <file.py> .

In command line, run:

```
$ python -i hello.py
"Hello World"
>>>
```

There are multiple ways to close the Python shell:

```
>>> exit()
```

or

```
>>> quit()
```

Alternatively, CTRL + D will close the shell and put you back on your terminal's command line.

If you want to cancel a command you're in the middle of typing and get back to a clean command prompt, while staying inside the Interpreter shell, use CTRL + C .

Try an interactive Python shell online .

Other Online Shells

Various websites provide online access to Python shells.

Online shells may be useful for the following purposes:

- Run a small code snippet from a machine which lacks python installation(smartphones, tablets etc).
- Learn or teach basic Python.
- Solve online judge problems.

Examples:

Disclaimer: documentation author(s) are not affiliated with any resources listed below.

- https://ideone.com/ -Widely used on the Net to illustrate code snippet behavior.
- https://repl.it/languages/python3 - Powerful and simple online compiler, IDE and interpreter. Code, compile, and run code in Python.
- https://www.tutorialspoint.com/execute_python_online.php - Full-featured UNIX shell, and a user-friendly project explorer.

Run commands as a string

Python can be passed arbitrary code as a string in the shell:

```
$ python -c 'print("Hello, World")'
Hello, World
```

This can be useful when concatenating the results of scripts together in the shell.

Shells and Beyond

Package Management - The PyPA recommended tool for installing Python packages is PIP . To install, on your command line execute `pip install <the package name>` . For instance, `pip install numpy` .

Shells - So far, we have discussed different ways to run code using Python's native interactive shell. Shells use Python's interpretive power for experimenting with code real-time. Alternative shells include IDLE - a pre-bundled GUI, IPython - known for extending the interactive experience, etc.

Programs - For long-term storage you can save content to .py files and edit/execute them as scripts or programs with external tools e.g. shell, IDEs (such as PyCharm), Jupyter notebooks , etc. Intermediate users may use these tools; however, the methods discussed here are sufficient for getting started.

Python tutor allows you to step through Python code so you can visualize how the program will flow, and helps you to understand where your program went wrong.

PEP8 defines guidelines for formatting Python code. Formatting code well is important so you can quickly read what the code does.

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. Python uses `=` to assign values to variables. There's no need to declare a variable in advance (or to assign a data type to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

```
a = 2
print(a)
# Out: 2

b = 9223372036854775807
print(b)
# Out: 9223372036854775807

pi = 3.14
print(pi)
# Out: 3.14

c = 'A'
print(c)
# Out: A

name = 'John Doe'
print(name)
# Out: John Doe

q = True
print(q)
# Out: True

x = None
print(x)
# Out: None
```

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 necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:

```
a = 2
print(type(a))
# Out: <type 'int'>

b = 9223372036854775807
print(type(b))
# Out: <type 'int'>

pi = 3.14
print(type(pi))
# Out: <type 'float'>

c = 'A'
print(type(c))
# Out: <type 'str'>

name = 'John Doe'
print(type(name))
# Out: <type 'str'>

q = True
print(type(q))
# Out: <type 'bool'>

x = None
print(type(x))
# Out: <type '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:

```
a, b, c = 1, 2, 3
print(a, b, c)
# Out: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

You can also assign a single value to several variables simultaneously.

```
a = b = c = 1
print(a, b, c)
# Out: 1 1 1
```

In this case, `a` , `b` , and `c` , are all independent -- changing one will not change the others:

```
a = b = c = 1
print(a, b, c)
# Out: 1 1 1
b = 2
print(a, b, c)
# Out: 1 2 1
```

However, note that changing the *contents* of an object that is referred to by a variable *will* be reflected through any other variables that reference the same object. That is, if `x` and `y` point at the same mutable object (e.g., *lists, dictionaries, sets, or byte arrays*), changing the contents of one *will* be seen through the other:

```
x = y = [7, 8, 9]   # x and y refer to the same list; i.e., they refer to same memory location
x = [13, 8, 9]      # now we are assigning a brand new list to x (memory location for x changed
print(y)            # y is unchanged, so it's OK
# Out: [7, 8, 9]
```

But:

```
x = y = [7, 8, 9]   # x and y refer to the same list i.e. refer to same memory location
x[0] = 13           # now we are replacing first element of x with 13 (memory location for x unc
print(y)            # this time y changed!
# Out: [13, 8, 9]
```

In plain English:

```
x = y = [7, 8, 9]   # points the names x and y at the same memory
x = y = 7           # creates two objects in memory, one named x the other named y
```

Nested lists are also valid in python. This means that a list can contain another list as an element.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
print x[2]
# Out: [3, 4, 5]
print x[2][1]
# Out: 4
```

Lastly, variables in Python do not have to stay the same type as which they were first defined -- you can simply use `=` to assign a new value to a variable, even if that value is of a different type.

```
a = 2
print(a)
# Out: 2

a = "New value"
print(a)
# Out: New value
```

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:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2             # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function())  # This line is OUTSIDE the function block
```

or

```
if a > b:             # If block starts here
    print(a)          # This is part of the if block
else:                 # else must be at the same level as if
    print(b)          # This line is part of the else block
```

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

```
if a > b: print(a)
else: print(b)
```

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

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

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:

```
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 ≥ 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.

Python 2.x ≤ 2.7

Python 2 allows mixing tabs and spaces in indentation; this is strongly discouraged. The tab character completes the previous indentation to be a multiple of **8** spaces . Since it is common that editors are configured to show tabs as multiple of **4** spaces, this can cause subtle bugs.

Citing PEP 8 :

> When invoking the Python 2 command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

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 semantics - at least in cases when (accidental) mixed indentation is possible. Editors 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)

Datatypes

Built-in Types

Booleans

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

```
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
```

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:

```
issubclass(bool, int) # True

isinstance(True, bool) # True
isinstance(False, bool) # True
```

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:

```
True + False == 1 # 1 + 0 == 1
True * True == 1 # 1 * 1 == 1
```

Numbers

- int : Integer number

  ```
  a = 2
  b = 100
  c = 123456789
  d = 38563846326424324
  ```

  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.

  ```
  a = 2.0
  b = 100.e0
  c = 123456789.e1
  ```

- complex : Complex numbers

  ```
  a = 2 + 1j
  b = 100 + 10j
  ```

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

Sequences and collections

Python differentiates between ordered sequences and unordered collections (such as set and dict).

- str : Character string; in Python 3 it is a **unicode** string, while in Python 2 it is a **byte string** .
- unicode : In Python 3 this type does not exist, str replaces it; in Python 2 it represents an **unicode encoded string** .
- bytes : In Python 3 this represents a **string of bytes** , without encoding defined; in Python 2 this is a synonym of str .
- tuple : An ordered collection of n values of any type (n >= 0); supports indexing; immutable; hashable if all its members are hashable.

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

- list : An ordered collection of n values (n >= 0); not hashable; mutable.

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

- set : An unordered collection of unique values.

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

- reversed : A reversed order of str with reversed function

```
a = reversed('hello')
```

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

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.

```
    a = {1: 'one',
         2: 'two'}

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

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.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x ≥3.0

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

Python 2.x ≤2.7

None is always less than any number (None < -32 evaluates to True`).

Testing the type of variables

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

```
a = '123'
print(type(a))
#Out: <class 'str'>
b = 123
print(type(b))
#Out: <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.

```
i = 7
if isinstance(i, int):
    i += 1
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

To test if something is of NoneType :

```
x = None
if x is None:
    print('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.

```
a = '123'
b = int(a)
```

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

```
a = '123.456'
b = float(a)
c = int(a)    # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)    # 123
```

You can also convert sequence or collection types

```
a = 'hello'
list(a) # ['h', 'e', 'l', 'l', 'o']
set(a)  # {'o', 'e', 'l', 'h'}
```

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 Python 3, 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

```
normal  = 'foo\nbar'   # foo
                       # bar
escaped = 'foo\\nbar'  # foo\nbar
raw     = r'foo\nbar'  # foo\nbar
```

Mutable and Immutable Data Types

In Python everything is an object. When define a variable like below

```
a = 5
```

We are creating an object 5 and tagging it with name a so that we can later use it with name a.

Now, we re-assign a like below

```
a = 6
```

Here, we are not modifying a rather creating a new object 6 and tagging it with a. So, it means in python we can never modify an int object rather create a new object and tag it with same name. So, int is an immutable data type.

Mutable data types are those in which we can modify the object.

Immutable Data Types::

- int , long , float , complex
- str
- byte
- tuple
- frozenset

Mutable Data Types::

Mutable Data Types:

- bytearray
- list
- set
- dict

You can use id() to check the identity of object, to verify whether same object is modified or new object is created.

```
>>> a=5
>>> id(a)    #154463584
>>> a=6
>>> id(a)    #154463572

>>> list1 = [1,2,3]
>>> id(list1)       # 3078128012L

>>> list1[0] = 5
>>> id(list1)       # 3078128012L

>>> list1.append(6)
>>> id(list1)       # 3078128012L
```

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.

Troubleshooting

- **Windows**

  If you're on Windows, the default command is python . If you receive a "'python' is not recognized" error, the most likely cause is that Python's location is not in your system's PATH environment variable. This can be accessed by right-clicking on 'My Computer' and selecting 'Properties' or by navigating to 'System' through 'Control Panel'. Click on 'Advanced system settings' and then 'Environment Variables...'. Edit the PATH variable to include the directory of your Python installation, as well as the Script folder (usually C:\Python27;C:\Python27\Scripts). This requires administrative privileges and may require a restart.

  When using multiple versions of Python on the same machine, a possible solution is to rename one of the python.exe files. For example, naming one version python27.exe would cause python27 to become the Python command for that version.

  You can also use the Python Launcher for Windows, which is available through the installer and comes by default. It allows you to select the version of Python to run by using py -[x.y] instead of python[x.y] . You can use the latest version of Python 2 by running scripts with py -2 and the latest version of Python 3 by running scripts with py -3 .

- **Debian/Ubuntu/MacOS**

  This section assumes that the location of the python executable has been added to the PATH environment variable.

  If you're on Debian/Ubuntu/MacOS, open the terminal and type python for Python 2.x or python3 for Python 3.x.

  Type which python to see which Python interpreter will be used.

- **Arch Linux**

  The default Python on Arch Linux (and descendants) is Python 3, so use python or python3 for Python 3.x and python2 for Python 2.x.

- **Other systems**

  Python 3 is sometimes bound to python instead of python3 . To use Python 2 on these systems where it is installed, you can use python2 .

Collection Types

There are a number of collection types in Python. While types such as int and 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. In Python, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, separated by commas, in square brackets:

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

A list can be empty:

```
empty_list = []
```

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

```
mixed_list = [1, 'abc', True, 2.34, None]
```

A list can contain another list as its element:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

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

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

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:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

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

```
names[0] = 'Ann'
print(names) # Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

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

Append object to end :- L.append(object) -> None

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names) # Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Add a new element to list on a specific index. L.insert(index, object) -- insert object before index

```
names.insert(1,"Nikki")
print(names)
['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia'] Outputs ['Ben', 'Alice', 'Bob', 'Crai
```

Remove the first occurrence of value:- L.remove(value) -> None

```
names.remove("Bob")
print(names) # Outputs ['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.

```
name.index("Alice")
0
```

Count length of list of list.

```
len(names)
6  #['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

count occurrence of any item in list

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

Reverse the list

```
a.reverse()
[4, 3, 2, 1, 1, 1]
#or
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Remove and return item at index (default last) :- L.pop([index]) -> item

```
names.pop() # Outputs 'Sia'
```

You can iterate over the list elements like below:

```
for element in my_list:
    print (element)
```

Tuples

A tuple is similar to a list except that is it 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:

```
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.

A tuple with only one member must be defined (note the comma) this way:

```
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:

```
state_capitals = {
    'Arkanasas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacremento',
    'Georgia': 'Atlanta'
}
```

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

```
ca_capital = state_capitals['California']
```

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

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

If dictionaries remind you of JSON, you're not dreaming! The native json module in the Python standard library converts between JSON and dictionaries. In fact, JSON is one of the most common uses of 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 :

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Or you can build a set using an existing list :

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

Check membership of the set using in :

```
if name in first_names:
    print(name)
```

You can iterate over a set exactly like a list, but remember: the values will be in a arbitrary, implementation-defined 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 doesnt need to be initialized everytime 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

```
>>> state_capitols = {
    'Arkanasas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacremento',
    'Georgia': 'Atlanta'
}
```

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

```
>>> state_capitols['Alabama']
Traceback (most recent call last):

  File "<ipython-input-61-236329695e6f>", line 1, in <module>
    state_capitols['Alabama']

KeyError: 'Alabama'
```

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

```
>>> from collections import defaultdict
>>> state_capitols = defaultdict(lambda: 'Boston')
```

What we did here is to set a default value (**Boston**) incase the give key doesnt exist. let us try to populate the dict as before

```
>>> state_capitols['Arkanasas'] = 'Little Rock'
>>> state_capitols['California'] = 'Sacremento'
>>> state_capitols['Colorado'] = 'Denver'
>>> state_capitols['Georgia'] = 'Atlanta'
```

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

```
>>> state_capitols['Alabama']
'Boston'
```

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

```
>>> state_capitols['Arkanasas']
'Little Rock'
```

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 2.x ≥ 2.3

```
name = raw_input("What is your name? ")
```

```
# Out: What is your name? _
```

Security Remark Do not use input() in Python2 - the entered text will be evaluated as if it were a Python expression (equivalent to eval(input()) in Python3), which might easily become a vulnerability. See this article for further information on the risks of using this function.

Python 3.x ≥ 3.0

```
name = input("What is your name? ")
# Out: What is your name? _
```

The remainder of this example will be using Python 3 syntax.

The function takes a string argument, which displays it as a prompt and returns a string. The above code provides a prompt, waiting for the user to input.

```
name = input("What is your name? ")
# Out: What is your name?
```

If the user types "Bob" and hits enter, the variable name will be assigned to the string "Bob" :

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name)
# Out: Bob
```

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:

```
x = input("Write a number:")
# Out: Write a number: 10
x / 2
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Out: 5.0
```

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 .

Built in Modules and Functions

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

```
>>> pow(2,3)    #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.

```
>>> dir(__builtins__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
```

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

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

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

```
>>> import math
>>> math.sqrt(16) # 4.0
```

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

```
>>>import math
>>>dir(math)

   ['__doc__', '__name__', '__package__', 'acos', 'acosh',
   'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
   'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
   'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
   'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
   'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
   'tan', 'tanh', 'trunc']
```

- 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()

```
>>> class MyClassObject(object):
...     pass
...
>>> dir(MyClassObject)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__'
```

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

```
>>> str(123)    # "123"
```

Creating a module

A module is an importable file containing definitions and statements.

A module can be created by creating a .py file.

```
# hello.py
def say_hello():
    print("Hello!")
```

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 the file that you are importing them into. (However, you can also put them into the Python lib directory with the pre-included modules, but should be avoided if possible.)

```
$ python
>>> import hello
>>> hello.say_hello()
=> "Hello!"
```

Modules can be imported by other modules.

```
# greet.py
import hello
hello.say_hello()
```

Specific functions of a module can be imported.

```
# greet.py
from hello import say_hello
say_hello()
```

Modules can be aliased.

```
# greet.py
import hello as ai
ai.say_hello()
```

A module can be stand-alone runnable script.

```
# run_hello.py
if __name__ == '__main__':
    import hello
```

Run it!

```
$ python run_hello.py
=> "Hello!"
```

If module inside a directory and need to detect by python, directory should contain a file called `__init__.py` .

String function - str() and repr()

There are two functions that can be used to obtain a readable 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:

```
s = """w'o"w"""
repr(s) # Output: '\'w\\\'o"w\''
str(s)  # Output: 'w\'o"w'
eval(str(s)) == s  # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Example 2:

```
import datetime
today = datetime.datetime.now()
str(today)  # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

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

```
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:

```
r = Represent(1, "Hopper")
print(r)  # prints __str__
print(r.__repr__)  # prints __repr__: '<bound method Represent.__repr__ of Represent(x=1,y="Hopp
rep = r.__repr__()  # sets the execution of __repr__ to a new variable
print(rep)  # prints 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # evaluates rep
print(r2)  # prints __str__ from new object
print(r2 == r)  # prints 'False' because they are different objects
```

Installation of Python 2.7.x and 3.x

Note : Following instructions are written for Python 2.7 (unless specified): instructions for Python 3.x are similar.

WINDOWS

First, download the latest version of Python 2.7 from the official Website (https://www.python.org/downloads/) . Version is provided as an MSI package. To install it manually, just double-click the file.

By default, Python installs to a directory:

```
 C:\Python27\
```

Warning: installation does not automatically modify the PATH environment variable.

Assuming that your Python installation is in C:\Python27, add this to your PATH:

```
C:\Python27\;C:\Python27\Scripts\
```

Now to check if Python installation is valid write in cmd:

```
python --version
```

LINUX

The latest versions of CentOS, Fedora, Redhat Enterprise (RHEL) and Ubuntu come with Python 2.7.

To install Python 2.7 on linux manually, just do the following in terminal:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
sudo make install
```

Also add the path of new python in PATH environment variable. If new python is in /root/python-2.7.X then run export PATH = $PATH:/root/python-2.7.X

Now to check if Python installation is valid write in terminal:

```
python --version
```

macOS

As we speak, macOS comes installed with Python 2.7.10, but this version is outdated and slightly modified from the regular Python.

> The version of Python that ships with OS X is great for learning but it's not good for development. The version shipped with OS X may be out of date from the official current Python release, which is considered the stable production version. (source)

Install Homebrew :

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install
```

Install Python 2.7:

```
brew install python
```

For Python 3.x, use the command `brew install python3` instead.

Installing external modules using pip

`pip` is your friend when you need to install any package from the plethora of choices available at the python package index (PyPI). `pip` is already installed if you're using Python 2 >= 2.7.9 or Python 3 >= 3.4 downloaded from python.org. For computers running Linux or another *Nix with a native package manager, `pip` must often be manually installed.

On instances with both Python 2 and Python 3 installed, `pip` often refers to Python 2 and `pip3` to Python 3. Using `pip` will only install packages for Python 2 and `pip3` will only install packages for Python 3.

Searching for a package is as simple as typing

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

Installing a package is as simple as typing

`$ pip install [package_name] # latest version of the package`

`$ pip install [package_name]==x.x.x # specific version of the package`

`$ pip install '[package_name]>=x.x.x' # minimum version of the package`

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

When your server is behind proxy, you can install package by using below command:

`pip --proxy http://<server address>:<port> install`

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

On Linux or OS X: `pip install -U pip`

You may need to use `sudo` with pip on some Linux Systems

On Windows: `py -m pip install -U pip` or `python -m pip install -U pip`

For more information regarding pip do read here .

Help Utility

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

```
>>> help()
```

You will receive information by entering keywords directly:

```
>>> help(help)
```

or within the utility:

```
help> help
```

which will show an explanation:

```
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
 |  Define the builtin 'help'.
 |
 |  This is a wrapper around pydoc.help that provides a helpful message
 |  when 'help' is typed at the Python interactive prompt.
 |
 |  Calling help() at the Python prompt starts an interactive help session.
 |  Calling help(thing) prints help for the python object 'thing'.
 |
 |  Methods defined here:
 |
 |  __call__(self, *args, **kwds)
 |
 |  __repr__(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

You can also request subclasses of modules:

```
help(pymysql.connections)
```

Close the helper with 'quit'

Remarks

python™

Python is a widely used programming language. It is:

- **High-level** : Python automates low level operations such as memory management. It leaves the programmer with a bit less control, but has many benefits including code readability, and minimal code expressions.
- **General-purpose** : Python is built to be used in all contexts and environments. An example for a non-general-purpose language is PHP: it is designed specifically as a server-side web-development scripting language. In contrast, Python *can* be used for server-side web-development, but also for building desktop applications.
- **Dynamically typed** : Every variable in Python can reference any type of data. A single expression may evaluate to data of different types at different times. Due to that, the following code is possible:

  ```
  if something:
      x = 1
  else:
      x = 'this is a string'
  print(x)
  ```

- **Strongly typed** : During program execution, you are not allowed to do anything that's incompatible with the type of data you're working with. For example, there are no hidden conversions from strings to numbers; a string made out of digits will never be treated as a number unless you convert it explicitly:

  ```
  1 + '1'  # raises an error
  1 + int('1')  # results with 2
  ```

- **Beginner friendly :)** : Python's syntax and structure is very intuitive. It is high level and provides constructs intended to enable writing clear programs on both a small and large scale. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It has a large and comprehensive standard library and many 3rd party easy to install libraries.

Its design principles are outlined in *The Zen of Python* .

Currently, there are two major release branches of Python which have some significant differences. Python 2.x is the legacy version, although it still sees widespread use. Python 3.x makes a set of backwards-incompatible changes which aim to reduce feature duplication. For help deciding which version is best for you, see this article .

The official Python documentation is also a comprehensive and useful resource, containing documentation for all versions of Python as well as tutorials to help get you started.

There is one official implementation of the language supplied by Python.org, generally referred to as CPython, and several alternative implementations of the language on other runtime platforms. These include IronPython (running Python on the .NET platform), Jython (on the Java runtime) and PyPy (implementing Python in a subset of itself).

Versions

Python 3.x

Version	Release Date
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

Version	Release Date
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