Python Cheatsheet

Version

Use Python 3! While Python 2 is still widely used, Python 3 is more logical, more consistent, and uses lots of modern concepts like unicode by default. Also note that Python 2 will "retire" in 2020.

Editor

Since the editor is your main tool for coding, you should choose it carefully. Some good multiplatform text editors are given below. All of them provide plugins for various applications like linters and can be used as a full-fledged Python IDE:

- Sublime Text — Extremely fast and extensible text editor written in C++ with an infinite trial period.
- Visual Studio Code — Reasonably fast and feature-rich text editor written in JavaScript by Microsoft.
- Atom — Reasonably fast and extensible text editor written in JavaScript by GitHub.

Since contexts in Python are defined by indentation, leading whitespace is part of Python's syntax! While this makes Python quite readable and forces you to write structured code, your editor should be set to replace tabs by spaces to prevent errors and confusion.

Packages

Packages can be imported with the `import` statement:

```
>>> import json
>>> import matplotlib.pyplot as plt
>>> from numpy import array
```

Some useful packages from the Python Standard Library:

- `collections` — Container datatypes
- `itertools` — Functions creating iterators for efficient looping
- `copy` — Shallow and deep copy operations
- `json` — JSON encoder and decoder
- `pickle` — Python object serialization
- `os.path` — Common pathname manipulations
- `pdb` — The Python Debugger

Third-party packages can be installed locally via the Package Installer for Python (pip) in a console:

```
$ pip install --user ...
```

- `ipython` — Architecture for interactive computing
- `numpy` — Tools for fast numerical computing
- `scipy` — Algorithms for scientific computing
- `sympy` — Symbolic calculation
- `matplotlib` — Versatile plotting library

Basics

Python is interpreted. In contrast to compiled software, this makes Python code less performant but allows for using and testing Python code interactively via interpreter sessions (`ipython` provides an improved interactive shell):

```
$ python
$ ipython
```

The Python interpreter stores the last expression value into the special variable "_". In Python scripts "_" can be used as an ordinary variable name (usually for temporary or insignificant values):

```
>>> def func():
>>>   return "important", "temporary", "insignificant"
>>> a, _, _ = func()
>>> a # "important"
```

Everything in Python is an object, including functions and classes. They can be stored in variables, lists, and dictionaries, which are objects themselves:

```
>>> def add(a, b):
>>>   return a + b
>>> lst = [42, "Hello World!", add]
>>> lst[2](2, 3) # 5
```

Data types

- Dictionaries are the central, highly optimized data structure in Python that maps arbitrary elements to keys, which can be of any hashable (immutable) type. Basically dictionaries are similar to phone books:

  ```
  >>> phonebook = {
  >>>   "Alice": 3141,
  >>>   "Bob": 2718,
  >>> }
  >>> phonebook["Alice"] # 3141
  >>> phonebook.keys() # dict_keys(['Alice', 'Bob'])
  ```

  Since dictionaries are in general unordered, you may wish to use `collections.OrderedDict` in some cases:

  ```
  >>> from collections import OrderedDict
  >>> d = OrderedDict(one=1, two=2)
  >>> d # OrderedDict([('one', 1), ('two', 2)])
  >>> d["two"] # 2
  ```

- Lists are dynamic, ordered data structures which can hold arbitrary elements:

  ```
  >>> lst = [1, "two"]
  >>> lst[0] = "one" # ["one", "two"]
  >>> del lst[1] # ["one"]
  >>> lst.append({"three": 3}) # ["one", {"three": 3}]
  ```

- Tuples are similar to lists but are immutable:

  ```
  >>> tpl = (1, "two")
  >>> tpl[0] = "one" # TypeError
  >>> del tpl[1] # TypeError
  >>> tpl.append(3) # AttributeError
  >>> list(tpl) # [1, 'two']
  ```

- `collections.namedtuple` provides tuples, which can be accessed by attributes:

  ```
  >>> from collections import namedtuple
  >>> Car = namedtuple("Car", "color mileage")
  >>> my_car = Car("red", 3.14)
  >>> my_car.color # "red"
  >>> my_car # Car(color="red" , mileage=3.14)
  ```

- `numpy.ndarray` is a fast array implementation for numerical calculations provided by the third-party package `numpy`:

  ```
  >>> import numpy as np
  >>> arr = np.array([3.141, 2.718])
  >>> arr # array([3.141, 2.718])
  ```

- Sets are immutable, unordered collections of unique, hashable objects:

  ```
  >>> s = set([1, 2, 2, 3])
  >>> s # {1, 2, 3}
  >>> len(s) # 3
  >>> s[0] # TypeError
  ```

Quick reference

				access by:
`dict`	{}	mutable	unordered	key
`OrderedDict`		mutable	ordered	key
`list`	[]	mutable	ordered	index
`tuple`	()	immutable	ordered	index
`namedtuple`		immutable	ordered	attribute
`set`		immutable	unordered	

Input and output

Python's `open` and `save` statements can be used for basic reading and writing of files. It is recommended to use them in combination with the `with` statement for context management:

```
>>> with open("file.txt", "w") as f:
>>>   f.write("Hello World!")
>>> # f.close() called on leaving the context
```

The `json` module allows for easy saving and loading of human-readable JSON files, which closely resemble Python's dictionary and list syntax:

```
>>> import json
>>> dct = {"one": 1, "list": [2, 3]}
>>> with open("dct.json", "w") as f:
>>>   json.dump(dct, f)
>>> with open("dct.json") as f:
>>>   dct_ = json.load(f)
>>> dct_ # {'one': 1, 'list': [2, 3]}
```

Python's `pickle` module allows for saving and loading whole objects into byte strings, which can be easily stored in files:

```
>>> import pickle
>>> class Obj:
>>>   attr = "A class attribute"
>>> pickle.dumps(Obj) # b'\x80\x03c__main__\nObj\nq\x00.'
```

Error handling

Any errors within a `try` block can be catched by an `except` statement. It is however wise to except only specific error types:

```
>>> try:
>>>   raise Exception("A wild Error occurred!")
>>> except Exception as e:
>>>   print("Exception:", e) # Exception: A wild ...
>>> except:
>>>   print("Any Error occurred!")
```

Debugging

Python's `assert` statement provides a basic debugging tool to test conditions:

```
>>> def apply_discount(price, discount):
>>>   new_price = price*(1 - discount)
>>>   assert 0 <= new_price <= price
>>>   return new_price
>>> apply_discount(7.99, 0.2) # 6.392
>>> apply_discount(7.99, 1.2) # AssertionError
```

The class `pdb` provides more sophisticated debugging abilities. It can be used to execute a script in debugging mode:

```
$ python -m pdb ...
```

The interactive debugger shell can evaluate python expressions like `print()`, which can be used to inspect variables. Some useful debugger commands are:

`n`	Execute the current line
`b`	Sets a breakpoint in the current line
`c`	Continue to the next breakpoint
`q`	Quit the debugger shell
`h`	Print a list of available commands

One can also set breakpoints manually in the Python script:

```
>>> import pdb; pdb.set_trace()
```

Tricks

In the following you find a selection of tricks, tips, and secrets inspired by the Python Tricks series.

Dictionaries

- Merging (Python 3.5+):

```
>>> x = {"a": 1, "b": 2}
>>> y = {"c": 3}
>>> z = {**x, **y} # {"a": 1, "b": 2, "c": 3}
```

- Sorting:

```
>>> dic = {"a": 3, "b": 1, "c": 2}
>>> sort = sorted(dic.items(), key=lambda x: x[1])
>>> sort # [("b", 1), ("c", 2), ("a", 3)]
```

- `get()` method:

```
>>> users = {42: "Alice", 1337: "Bob"}
>>> def greeting(userid):
>>>   return("Hi {}!"
>>>          .format(users.get(userid, "there")))
>>> greeting(42) # "Hi Alice!"
>>> greeting(1998) # "Hi there!"
```

Lists

- List comprehension:

```
>>> even_squares = [x**2 for x in range(10)
>>>                 if not x % 2]
>>> even_squares # [0, 4, 16, 36, 64]
```

- List slicing:

```
>>> lst = list(range(1, 7))
>>> lst # [1, 2, 3, 4, 5, 6]
>>> lst[1:-2] # [2, 3, 4]
>>> lst[1:-2:2] # [2, 4]
>>> lst[::-1] # [6, 5, 4, 3, 2, 1]
```

Variables

- In-place value swapping:

```
>>> a = 42
>>> b = 1337
>>> a, b = b, a
>>> a, b # (1337, 42)
```

Objects

- `__repr__` and `__str__` dunder methods:

```
>>> import datetime
>>> today = datetime.date.today()
>>> today # datetime.date(2019, 4, 17)
>>> str(today) # '2019-04-17'
>>> repr(today) # 'datetime.date(2019, 4, 17)'
```

- `__dict__` contains an attributes like variables and functions:

```
>>> class Obj:
>>>   def method(self): print(self)
>>> obj = Obj()
>>> obj.__class__ # __main__.Obj
>>> obj.__class__.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'method': <function ...>,
              '__dict__': <attribute ...>,
              '__weakref__': <attribute ...>,
              '__doc__': None})
```

Functions

- Lambda functions:

```
>>> add = lambda x, y: x + y
>>> add(5, 3) # 8
>>> (lambda x, y: x + y)(5, 3) # 8
```

- Function argument unpacking:

```
>>> def func(x, y, z):
>>>   print(x, y, z)
>>> tup = (1, 0, 1)
>>> dic = {"x": 1, "z": 1, "y": 0}
>>> func(*tup) # 1, 0, 1
>>> func(**dic) # 1, 0, 1
```

- Keyword-only function arguments:

```
>>> def func(a, b, *, c=None):
>>>   return "Hello!"
>>> func(1, 2, 3) # TypeError
>>> func(1, 2, c=3) # 'Hello!'
```

Formatting

- Format String Syntax:

```
>>> "{:.2f} Euro".format(3.141) # '3.14 Euro'
>>> "{c.imag:.0f}i".format(c=(1+2j)) # '2i'
>>> "{1} -> {0}".format("a", "b") # 'b -> a'
```

- Formatted string literals (Python 3.6+):

```
>>> answer = 42
>>> str = f"The answer is {answer}"
>>> str # 'The answer is 42'
```

- Formatted strings from dictionaries with `json.dumps`:

```
>>> import json
>>> dct = {"b": 1, "a": 2}
>>> print(json.dumps(dct, indent=2, sort_keys=True))
```

```
{
  "a": 2,
  "b": 1
}
```

Copying

- References, shallow and deep copies:

```
>>> from copy import deepcopy
>>> lst = [1, [2, 3]]
>>> reference = lst
>>> shallow_copy = lst[:]
>>> deep_copy = deepcopy(lst)
>>> lst[0] = "a"
>>> lst[1][1] = 4
>>> reference # ['a', [2, 4]]
>>> shallow_copy # [1, [2, 4]]
>>> deep_copy # [1, [2, 3]]
```

Boolean expressions

- Comparisons (`is` vs `==`):

```
>>> obj = [1, 2, 3]
>>> ref = obj
>>> obj is ref, obj == ref # (True, True)
>>> copy = obj[:]
>>> obj is copy, obj == copy # (False, True)
```

- Test multiple flags with `any()` and `all()`:

```
>>> votes = [True, False, False]
>>> any(votes) # True
>>> all(votes) # False
```

Tools

- Better tracebacks with `faulthandler` (Python 3.3+):

```
>>> import faulthandler
>>> faulthandler.enable()
```

- Measure execution times with `timeit`:

```
>>> from timeit import timeit
>>> cmd = "'-'.join(str(n) for n in range(100))"
>>> time = timeit(cmd, number=100)
>>> time # 0.0020432909950613976
```

- Find the most common elements with `Counter`:

```
>>> from collections import Counter
>>> counter = Counter("Hello World!")
>>> counter.most_common(2) # [('l', 3), ('o', 2)]
```

References

- Python documentation — The Python documentation with lots of examples
- PEP 8 — Style guide for Python code
- Real Python — Guides, tutorials, and news about Python