PYTHON CHEAT SHEET

Welcome to DevelopMentor's Python quick start booklet. We wrote this for all beginning to intermediate Python developers to keep close to their editors and keyboards.

We have sample use-cases for many of the most common language constructs and tasks. In addition to language constructs, we have also added many common tasks you might need to perform such as reading a file or working with a database.

To use this booklet, just find the task or language topic you need a quick reminder on in the table of contents and jump to that section.

We hope you find the booklet useful. If you want more in-depth Python training check out our classes and webcasts.

Topics:

Language

This section covers the language fundamentals such as how to create variables, functions, loops, and classes.

Variables

Variables are declared without specifying the type.

```
name = 'DevelopMentor'
n1 = 4
n2 = 38
ans = n1 + n2
```

Conditionals

Conditional expressions make heavy use of Python's truthiness inherent in objects. Python defines the following to be **`False`**:

- `None`
- `False`
- zero of any numeric type, for example, `0`, `0L`, `0.0`, `0j`.
- any empty sequence, for example, `''`, `()`, `[]`.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, which returns False`

Everything else is **`True`**.

Conditionals in Python can operate on any type. Fundamentally we have two boolean values `True` and `False`. Additionally, any *truthy* statement (see above) can also be involved in such tests.

back to table of contents

Conditional statements can be used in isolation:

```
operations  [...]
is_active = read_config(‘isactive’)
should_run = operations and is_active
# should_run is True or False
```

Truthy and *falsey* elements are combined with the `and` and `or` keywords as well as negated with the `not` keyword.

```
operations  [...]
is_active = read_config(‘isactive’)
is_disabled = not operations or not is_active
# is_disabled is True or False
```

Usually, these are combined with if statements or loops.

if statements

```
operations  [...]
is_active = read_config(‘isactive’)

if is_active and opreations:
    run_batch_job()
```

if statements can have alternative clauses via else if and else:

```
operations  [...]
is_active = read_config(‘isactive’)

if is_active and operations:
    run_batch_job()
elif not operations:
    print(‘nothing to do!’)
else
    print(‘not running, disabled’)
```

back to table of contents

Loops

Python has several types of loops, *for-in* and *while*, along with other language elements that accomplish looping without being so explicit (e.g. *map or list comprehensions*).

While loops are closely related to conditionals. They take the form:

```
while BOOLEAN:
    block
```

So we might process our operations from the prior example as follows:

```
operations  [...]
is_active = read_config('isactive')

while operations and is_active:
    op = operations.pop()
    process(op)
```

The most popular type of loop is the for-in loop and it can take several *forms*. The most common loop takes an iterable set and loops across it.

```
items = ['a', 'b', 'c']
for i in items:
    print('The item is {}'.format(i))
```

It can be combined with range to simulate a numerical for loop:

```
for i in range(1,10):
    item = getItemByIndes(i)
    print('The item is {}'.format(item))
```

back to table of contents

Functions

Python functions are defined using the `def` keyword. Neither the return type or argument types are specified. A typical method looks like:

```
def print_header():
    print(“---------------------------”)
    print(“ sample app pro v1.1”)
    print(“---------------------------”)
```

Methods can also return values.

```
def find_user_age():
    text = input(“What is your age in years? “)
    age = int(txt)

    return age
```

as well as take value (including default values).

```
def configure_app(server_ip, mode = “Production”):
    # work with server_ip and mode

configure_app(‘1.2.3.4’) # use default
configure_app(‘127.0.0.1’, mode=”DEV”) # replace default
```

back to table of contents

Classes

Python has rich support for classes. Create them as:

```
class Pet(Animal):
    def __init__(self, name):
        self.name = name
    def play():
        pass
```

The pet class derives from Animal (not shown).

Create the pet as:

```
fluffy = Pet(‘Fluffy the cat’)
```

back to table of contents

Ecosystem

This section covers setting up the Python environment and installing external packages.

The biggest decision you have to make / identify is whether your app is using Python 3 or Python 2.

On OS X or Linux, Python versions can be selected by name:

```
> python2 # python 2
> python3 # python 3
```

On Windows you'll need the proper path:

```
c:\python27\python.exe # python 2
c:\python34\python.exe # python 3
```

Packaging

Packages are managed via pip.exe. Pop comes with Python 3.3 when installed. It must be added to older versions of Python.

External packages can be found on PyPI:

https://pypi.python.org/pypi

There are *many* good packages there. If you want to install one (e.g. requests), you simply use pop as:

```
pip install requests
pop3 install requests
```

Pip3 only works on OS X / Linux with Python 3 installed.

back to table of contents

Imports

Code uses modules (builtin and external) via the import keyword:

```
import requests

r = requests.get(‘https://develop.com’)
```

Virtual Environments

If you wish to isolate your Python environment, you can install and use virtualenv.

```
> pip install virtualenv
```

Then create a virtual environment using:

```
> virtualenv ./my_env
```

And activate it via

```
> cd my_env/bin
> .activate
```

Now all Python commands target this environment. (Note: Windows is slightly different.)

back to table of contents

Common Tasks

Open File

File IO uses the open method. The *with* context manager is recommended to close the file as soon as possible.

```
csvFileName = “SomeData.csv”

with open(csvFileName, ‘r’, encoding=”utf-8”) as fin:
    for line in fin:
        print(line, end=’’)
```

Make Type Iterable

Custom classes add iteration by implementing the `__iter__` magic method. The quickest way to do this is via the `yield` keyword.

```
class Cart:
    def__init__(self):
        self.__items = {}

    def add(self, cartItem):
        self.__items.append(cartItem)

    def__iter__(self):
        for item in self.__items:
            yield item
```

back to table of contents

Then this shopping cart can be iterated:

```
cart = Cart()
cart.add(CartItem('Tesla', 63000))
card.add(CartItem('BMW', 42000))

total = 0
for item in cart:
    total += item.price

# total is 105,000
```

Convert Field to Property

Sometimes a field should be read-only or computed when accessed. Other times, you want to add validation to the setting of a field. For these cases, a **property** is ideal.

Here is how we create a read-only or computed property (note the __field makes it private which is part of the encapsulation).

```
class Cart:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

    @property
    def total(self):
        total_value = 0
        for i in self.items:
            total_value += i.price
        return total_value
```

back to table of contents

And to use the cart, we treat total as a field (but the method is invoked as a get property).

```
cart = Cart()
cart.add(Item(name='CD', price=12.99))

print("The total for the order is {}".format(cart.total))
```

Read From a Database

A very common use of Python is data access. You can connect to most types of databases (SQLite, MySQl, SQL Server, etc.) using the DBAPI2.

Her is a simple connect and query some data from a SQLite database.

```
import sqlite3
import os

file = os.path.join('.', 'data', "sample_data.sqlite_db")
conn = sqlite3.connect(file)
conn.row_factory = sqlite3.Row

search_text = 'Rise'
sql = 'SELECT id, age, title FROM books WHERE title like?'
cursor = conn.execute(sql, ('%'+seach_text+'%',))
for r in cursor:
    print(r, dict(r))
    print(r['id'], r['title'], type(r))
```

back to table of contents

Create List

Working with lists is one of the most common collection type operations you do in Python. Here are a few things you can do with lists.

```
# create an empty list (two ways)
data = []
data = list()

# create a list with data
data = [1,1,2,3,5,8]

# access items
third = data[2] #zero based

# sort the data
data.sort(key=lambda n: -n)
# data = [8,5,3,2,1,1]
```

back to table of contents

Create a List Comprehension

List comprehensions and generator expressions are condensed versions of procedural for-in loops and generator methods. List comprehensions are statements within square braces which are reminiscent of standard list literals (last =[1,2,7]):

```
numbers = [1,1,2,3,5,8,13,21]

even_squares = [
    n*n
    for n in umbers
    if n % 2 == 0
]

# even_squares = [4, 64]
```

Generator expressions are *very* similar, but use parentheses instead of square braces.

```
numbers = [1,1,2,3,5,8,13,21]

even_squares = (
    n*n
    for n in numbers
    if n % 2 == 0
)

for s in even_squares:
    print(s)

# prints: 4, then 64
```

back to table of contents

Parsing JSON

JSON is a very popular and concise storage format. It looks roughly like this:

```
{
    “property”: value,
    “property”: value,
    “property”: {
        “sub_prop1”: value,
        “sub_prop2”: value
    }
    “property”: [value, value, value]
}
```

It is comprised of name / value pairs as well as other JSON objects and arrays.

We can read and parse JSON in Python using the builtin JSON module:

```
import json

jsonTxt = ‘{“name”: “Jeff”, “age”: 24}’

d = json.loads(jsonTxt)
print( type(d) )
print( d )

# prints:
# <class ‘dict’>
# {‘age’: 24, ‘name’: ‘Jeff’}
```

Similarly, we can take Python dictionaries and turn them into JSON via:

```
json.dumps(d)
# returns: {“age”: 24, “name”: “Jeff”}
```

back to table of contents

Parsing XML

Python has builtin XML capabilities. To see them in action, consider the following RSS feed (an XML document):

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
    <channel>
        <title>Michael Kennedy on Technology</title>
        <link>http://blog.michaelkennedy.net</link>
        <item>
            <Title>Watch Building beautiful web...</title>
            <link>http://blog.michaelkennedy.net/...</link>
        </item>
        <item>
            <Title>MongoDB for .NET developers</title>
            <link>http://blog.michaelkennedy.net/...</link>
        </item>
        <item>...</item>
    </channel>
</rss>
```

We can use the xml.etree module to find all the blog posts:

```
from xml.etree import ElementTree
dom = ElementTree.parse("blog.rss.xml")

items = dom.findall('channel/item')
print("Found {0} blog entries.".format(len(items)))

entries = []
for item in items:
    title = item.find('title').text
    link  = item.find('link').text
    entries.append( (title, link) )
```

back to table of contents

Then entries contains:

```
Found 50 blog entries.
entries[:3] =>
[
    ('Watch Building beautiful web...', 'link1'),
    ('MongoDB for .NET developers', 'link2').
    ('...', 'link3'),
]
```

back to table of contents