Python Supplement

August 31, 2024

Ramasamy Kandasamy

1. Core

`help()`	Help about a module or function. Eg: `import numpy as np` Eg: `help(np)` Eg: `help(np.sort)`
`.__name__`	Name of an object, function, module, etc. Eg: `foo.__name__ # Returns 'foo'.` Eg: `f = foo` Eg: `f.__name__ # Returns 'foo'.`
`id(x)`	Gives the address of the variable `x`. Can be used to check if two variables point to the same object. The address cannot be derefenced like a C pointer.
`isinstance()`	Eg: `isinstance(x, str)`, returns true if x is a string. `isinstance(x, (str, int))`, returns true if x is either a string or an integer.
`del()`	Delete objects.
`globals()`	Gives dictionary representing the global symbol table.
`.get()`	Get object by name. Eg: `bar = globals.get('foo')`.
`global`	Declares and object as a global variable. Eg: `global foo`. But, `global foo = 2`, does not work. Assignment during global declaration does not work. Also see list comprehension.
*	Unpacks a sequence like a list or tuple to function arguments. Eg: `l = [1, 2, 3]` Eg: `foo(*l)` Eg: `a, *b, c = [1, 2, 3, 4, 5] # b = [2, 3, 4]`
**	Upacks a dictionary into keyword arguments. Eg: `kwargs = 'a': 1,'b' = 2.` Eg: `foo(**kwargs)` Inside function foo `a` and `b` becomes variables with values `1` and `2` respectively.
`zip`	Zip two iterables. Eg: `x = [1, 2]`. Eg: `y = [3, 4]`. Eg: `w = list(zip(x, y))`. Eg: `w = [(1, 3), (2, 4)]`.
`enumerate`	Enumerate iterables. Eg: `for id, x in enumerate(l):`. Eg: `for id, (x, y) in enumerate(zip(l1, l2)):`.
`eval`	Evaluate a string expressions. Eg: `eval('1 + 2')`. Returns 3.

Caution: **A note on circular imports.**
When two modules import from each other it causes unexpected behaviour. Strategies to avoid circular imports:

1. Refactor shared functionalities to a distinct module.
2. Use local imports.
3. Use lazy imports: `importlib.import_module()`.

NOTE: **Memory efficient assignment.**

```
x = [1, 2, 3]
x = [4, 5, 6] # x has different address.
x[:] = [4, 5, 6] # x has the same address.
```

List comprehensions

- **Basic Syntax.**
  `[f(x) for x in iterable]`. Eg: `[x ** 2 for x in range(4)]`
  `[0, 1, 4, 9]`
- **With conditions.**
  `[x ** 2 if x%2==0 else 2 * x for x in range(10)]`
  `[0, 2, 4, 6]`
- **Filtering items by condition.**
  `[x ** 2 for x in range(10) if x%2==0]`
  `[0, 4]`
- **Nested list.**
  `[(x, y) for x in [1, 2] for y in [3, 4]]`
  `[(1, 3), (1, 4), (2, 3), (2, 4)]`
- **Flattening a list.**
  `m = [[1, 2], [3, 4]]`
  `[x for v in m for x in v]`
  `[1, 2, 3, 4]`

Special Variables and Naming Conventions

`_foo`	Internal use. Not for public access.
`__foo`	Name mangling to prevent accidental overrides. Interpreter changes the name to prevent override in child class.
`__foo__`	Special variables that are part of python, don't create your own.
`foo_`	To avoid conflict with keywords.
`__file__`	Path to current script of file.
`__name__`	Name of the module or `"__main__"` if run directly. Eg: If the script is `foo.py` Then `__name__ == "foo"`.
`__version__`	Version of the module. Does not work for all (eg: os). Eg: `np.__version__`. Eg: `os.__version__`. This does not work.

TODO: Generators,

2. OS, Shutil, Sys, Etc.

os

`chdir()`	
`getcwd()`	
`listdir()`	
`mkdir()`	Create a directory at the path.
`makedirs()`	Recursively creates the directories. Like `mkdir -p` in bash.
`rmdir()`	To delete **empty** directories.
`rename()`	`os.rename(src, dst)`. Rename for files and directories.
`system()`	Execute system commands. Return the exit status. Eg: `exit_status = os.system('ls')`.

shutil

`rmtree()`	`shutil.rmtree('mydir')`.
`move()`	`shutil.move(src, des)`.
`copy()`	`shutil.copy(src, des)`. Copy single file without preserving meta data.
`copy2()`	Copy single file with preserving meta data.
`copytree()`	Copy entire directory and it's contents.

os.path

`isdir()`	
`isfile()`	
`exists()`	
`join()`	
`dirname()`	
`basename()`	
`abspath()`	Return the absolute path to a file. Eg: `os.path.abspath('foo')`.
`realpath()`	Real path Resolves any symlinks along the way.

sys

`argv`	Command line arguments as a list of strings.
`exit()`	Exit and return an exit status to the calling process.
`version`	Get python version.
`path`	Manage list of search paths for modules. `sys.path.append('foo')`. Add the directory `foo` to the path.

CAUTION!!

In `sys.path.append()`, relative path is acceptable, but it is relative to the directory from where the script is being executed, not relative to where the script file is located.

Best practice:

```
script_dir = os.path.dirname(os.path.abspath(__file__))
sys.path.append(os.path.join(script_dir,\
relative/path/to/directory'))
```

3. Project setup, etc.

3.1. Virtual environments and package management

Virtual Environments

```
python3 -m venv myenv # Create an environment.
source myenv/bin/activate # Activate the environment.
deactivate # Deactivate the environment.
```

Pip

install	Eg: `pip3 install pkg`. `pip3 install pkg==<ver-num>`. Install specific version. `pip3 install --upgrade pkg`. `pip3 install -r requirements.txt`.
list	`pip3 list`.
uninstall	`pip3 uninstall pkg`.
freeze	`pip3 freeze > requirements.txt`

3.2. Documentation

Documentation using `sphinx`.

- Install: `pip3 install sphinx`.
- Initialize a docs directory: `sphinx-quickstart`.
  This will create `source` and `build` directories.
- `source/conf.py`: Setup configurations.
- `source/index.rst`: Starting point for the documentation.
- `source/modules.rst`: Add modules here.
- `make html`: Execute from the directory containing the makefile.
- `make clean`: To clean the existing build.

conf.py
The template is generated by `sphinx-quickstart`

```
<---Template by sphinx --->
# Add path.
import sys
sys.path.append(path/to/src)

# Configure extension.
extensions = [
    'sphinx.ext.autodoc'
    'sphinx.ext.autosummary'
    'sphinx.ext.mathjax'
    'sphinx.ext.napoleon'
    'sphinx.ext.viewcode'
]

# Configure theme.
# Default is alabaster.
# Install furo with pip.
html_theme = 'furo'
```

Example: `modules.rst`

```
src
===

.. toctree::
   :maxdepth:4

   myclass
```

3.3. Testing

Unit test using `unittest` module.
Test directory:

```
myrepo/
|-- test/
    |-- __init.py__    # Could be empty but necessary.
    |-- MyClassTest.py # Tests for MyClass.py.
```

Simple test file:

```
import unittest

class MyClassTest(unittest.TestCase):
    def setup(self):
    # Setup vars etc.

    def test_foo(self):
    # Write tests.
    self.assertAlmostEqual(obs, exp)

if __name__ == '__main__':
    unittest.main()
```

Using unittest:

- `$ python3 -m unittest discover`: Run all the tests.
- `$ python3 -m unittest myclasstest.py`: Run only `myclasstest.py`.
- `$coverage run -m unittest discover`: Run the test with coverge to get coverage report.
- `$ coverage report`: Prints coverage report to stdout.
- `$ coverage html`: Generates an html report.
  This is stored in `htmlcov` directory.

4. Useful Libraries

4.1. Serialization

pickle

`import pickle as pkl`

- Save an object:
  `pkl.dump(foo, open('foo.pkl', 'wb'))`.
- Load and object:
  `foo = pkl.load(open('foo.pkl', 'rb'))`.

yaml

Mostly used to store configuration files.
`import yaml`

- Read an YAML file.
  YAML files are loaded to a dictionary object.
  For most cases use `safe_loader`.
  ```
  with open('config.yaml', 'r') as file:
    config = yaml.safe_load(file)
  ```
  The following support wider capabilities, but is less secure:
  ```
  with open('config.yaml', 'r') as file:
     data = yaml.load(file, Loader=yaml.FullLoader)
  ```
- Save an YAML file:
  ```
  with open('config.yaml', 'r') as file:
    yaml.dump(yaml, file, default_flow_style=False)
  ```
  NOTE: Setting `default_flow_style` to `False` writes the output in block style and has better readability.

Examples of YAML files

```
# Comments
n: 10 # Integer.
x: 0.1 # Float.
s: "Hello, World!" # String.
list_1: [1, 2, 3] # 1D List.
list_2: [1, 2, 3
         4, 5, 6] # A list can span multiple lines.
mat:
    - [1, 2, 3]
    - [4, 5, 6]
    - [7, 8, 9] # A list of list.

inventory: # List of dictionaries.
    - id: 123
      desc: apple
    - id: 124
      desc: mango
```

```
description: |
  This is a multiline string.
  Newlines and indentation
  will be preserved.

  It can contain empty lines,
  special characters, etc.

description: >
  This is a multiline string
  that will be folded into a
  single line when loaded.

  Empty lines will be preserved
  as line breaks.
```

4.2. Copy

`import copy`

- `copy.copy()`. Creates a shallow copy.
  Creates a copy of the original object, but not of the objects contained within this original object.
- `copy.deepcopy()`. Creates new copy of the original object and all the objects within this original object.

4.3. Datetime

datetime.date

Represents date.

- `datetime.date(Y, M, D)`.
  Eg: `d = datetime.date(1903, 3, 14)`.
  Individual components:
  - `year = d.year`
  - `month = d.month`
  - `day = d.day`
- `datetime.today()`.

datetime.time

Represents time.

- `t = datetime.time(H, m, s)`.
  Eg: `t = datetime.time(23, 15, 17)`.
  - `hr = t.hour`
  - `min = t.minute`
  - `sec = t.second`

datetime.datetime

Represents both date and time.

- `today = datetime.datetime.today()`
- `now = datetime.datetime.now()`.
- `dt = datetime.combine(d, t)`.

datetime.timedelta

- `delta = datetime.timedelta(hours = 4)`
- `t2 = t1 + delta`
- `delta = t2 - t2`. `delta` is an instance of `timedelta`.

Etc.

- `d.strftime("%Y%m%d%H%M%S")`
- `d = datetime.strptime(date_string, "%Y%m%d%H%M%S")`
- `d.replace(year = 2025)`
- `d.weekday()`
  0 → Monday, 6 → Sunday.

Formats:

%Y	-	2023	%H	-	24H
%y	-	23	%I	-	12H
%m	-	01	%M	-	mins
%-m	-	1	%S	-	secs
%d	-	01	%f	-	μs
%-d	-	1	%p	-	AM/PM

%U	→	First sunday of the year is start of week 01. Days before that belong to week 00.
%W	→	First monday of the year is start of week 01. Days before that belong to week 00.
%V	→	Week number by ISO system.

Regular Expression

`import re`

Usage for finding patterns: `re.<method>(pattern, text)`

Methods in re.

`search`	Return the first match as a match object.
`match`	Match only at the beginning of the string. Returns a match object.
`findall`	Returns a list of all non-overlapping matches.
`finditer`	Returns an iterator to non-overlapping matches.
`sub`	Replace pattern. Usage: `re.sub(pattern, replacement, text)`.
`split`	Split by pattern. Eg: Split by white spaces. `pattern = r"\s+"` `re.split(pattern, text)`
`group`	Returns the entire match as string.
`group(n)`	Returns the n-th group, n=0 is same as `.group()`
`groups`	Returns a tuple of all the matching groups.

Patterns

`[]`	Match characters inside.
`.`	Match any char except newline.
∧	Match beginning of line. Inside [], it negates the pattern.
`$`	Match any char except newline.
`d`	Digits.
`D`	Any character other than digits.
`w`	Any alphanumeric.
`W`	Any character other than alphanumeric.
`s`	Any white space character: space, tab, linebreak.
`S`	any non-whitespace character.
`*`	0 or more repitition.
`+`	1 or more repitition.
`?`	0 or 1 repitition.
{n}	Exactly n repititions.
{m, n}	Between m and n repitions.
()	Group patterns.

```
pattern = r"(\d{4})-(\d{2})-(\d{2})"
text = "2024-08-24"

match = re.match(pattern, text)

if match:
    print(match.group())  # Output: '2024-08-24'.
    print(match.group(1)) # Output: '2024' (first group).
    print(match.group(2)) # Output: '08' (second group).
    print(match.group(3)) # Output: '24' (third group).
    print(match.groups()) # Output: ('2024', '08', '24').
```

5. OOP in Python

Classes and Instances

```
# Code-1
class Employee:
    raise_amount = 1.04 # 4% raise.
    num_of_emps = 0
    def __init__(self, first, last, salary):
        self.first = first
        self.last = last
        self.salary = salary
        Employee.num_of_emps += 1
    def fullname(self):
        return '{} {}'.format(first, last)
```

Properties of a class

- **Instance:** Eg: `e1 = Employee("Sam", "Gamgee", 100)`
  This creates an instance of the class `Employee` called `e1`. For example: `raise_amount` is not present in `e1.__dict__` but is present in `Employee.__dict__`
- **Attribute:** Data in the class.
  Eg: `first` is an attribute of the class `Employee`. This can be accessed as `e1.first`
  Class variables and **Instance variables**.
- **Method:** Functions defined in a class. In the above class `__init__` and `fullname` are methods of the class `Employee`. There are two access these:
  - `e1.fullname()`
  - `Employee.fullname(e1)`

Namespace

`__dict__` can be used to view the namespace of a class or its instance. Eg:

```
print(Employee.__dict__)
print(e1.__dict__)
print(e2.__dict__)
```

Class Variables and Instance Variables

In the above class `raise_amount` is a class variable. It is part of the namespace of the class and is not yet present in the namespace of the instance. An instance recieves it from the class when this variable in the instance is called.
It is important to distinguish between **class variable** and **instance variable**. If an instance variable has the same name as a class variable, then it supercedes the class variable.

Eg:

```
# Code-2
e1 = Employee("Sam", "Gamgee", 200)
e2 = Employee("Frodo", "Baggins", 100)

print(Employee.pay_raise)
print(e1.pay_raise)
print(e2.pay_raise)
print("---")

Employee.pay_raise = 1.5
print(Employee.pay_raise)
print(e1.pay_raise)
print(e2.pay_raise)
print("---")

e1.pay_raise = 2

'''
    Now e1.pay_raise is instance variable.
'''

print(Employee.pay_raise)
print(e1.pay_raise)
print(e2.pay_raise)
print("---")


Employee.pay_raise = 3
'''
This does not affect e1.pay_raise,
because it is now an instance variable.
However, e2.pay_raise still refers to class variable.
'''
print(Employee.pay_raise)
print(e1.pay_raise)
print(e2.pay_raise)
```

Output:

```
1.04
1.04
1.04
---
1.5
1.5
1.5
---
1.5
2
1.5
---
3
2
3
```

Methods

Three types of methods:

- **Regular methods**: Takes first argument as the instance.
- **Class methods**: Takes the first argumetn as the class.
  Defined using `@classmethod`
- **Static methods**: Does not take class or the instance as arguments.
  Defined using `@staticmethod`

Examples:

```
# Code-3
class Employee:
    # Add __init__ and fulname functions.
    @classmethod
    def from_string(cls, s):
        first, second, pay = s.split('-')
        return cls(first, second, pay)

    @staticmethod
    def is_workday(day):
        if(day.weekday() == 5 or day.weekday() == 6):
            return False
        return True

e1 = Employee.from_string('Sam-Gamgee-100')
print(e1.fullname())

import datetime
my_date = datetime.date(2023, 2, 18)
print(e1.is_workday(my_date))
```

Output:

```
Sam Gamgee
False
```

Inheritance

```
class Developer(Employer):
    pass

dev_1 = Developer('Sam', 'Gamgee', 100)
```

Here `dev_1` behaves exactly like an instance of the class `Employer`. Inherited class can use the constructor of parent class as follows.

```
class Developer(Employee):
    def __init__(self, first last, pay, prog_lang):
        super().__init__(first, last, pay)
        self.prog_lang = prog_lang
```

A second way to use parent constructor is as follows:

```
class Developer(Employee):
    def __init__(self, first, last, pay, prog_lang)
    Employee.__init__(self, first, last, pay)
    self.prog_lang = prog_lang
```

This second method is necessary for multiple inheritance. However using `super()` is better for single inheritance.

isinstance and issubclass

- `isinstance` Tests if something is an instance of a class.
- `issubclass` Tests if a class inhertis from another.

Eg:

```
print(isinstance(emp1, Employee))
print(isinstance(emp1, Developer))
print(isinstance(dev1, Employee))
print(isinstance(dev1, Developer))
print(issubclass(Developer, Employee))
print(issubclass(Employee, Developer))
```

```
True
False
True
True
True
False
```

Special Methods

The methods with double undescore around them are called **Dunder methods**, for example `__init__`.

Two important dunder methods

- `__repr__`: The goal of repr is to be as unambiguous as possible.
- `__str__`: The goal of str is to be readable.

Eg:

```
class Employee:
    # Define all other functions.
    def __repr__(self):
        return "Employee('{}', '{}', {})".\
            format(self.first, self.last, self.pay)

    def __str__(self):
        return "{} {}: {}".\
            format(self.first, self.last, self.email)


print(emp1)
```

```
Sam Gamgee: Sam.Gamgee@email.com
```

Without `str` print will default to `repr`. The output would be:

```
Employee('Sam', 'Gamgee', 100)
```

Operator overloading

Dunder methods can be used to overload operators.
Examples.

- `__get_item__ []`.
- `__add__` defines addition operator.
- `__len__` defines `len()`.

Property Decorators

Example:

```
class Employee:
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay

    @property
    def fullname(self):
        return '{} {}'.format(self.first, self.last)

    @fullname.setter
    def fullname(self, name):
        first, last = name.split(' ')
        self.first = first
        self.last = last

    @fullname.deleter
    def fullname(self):
        self.first = None
        self.last = None

emp1 = Employee('Sam', 'Gamgee', '100')
print(emp1.fullname)

# NOTE: We this is not emp1.fullname()
emp1.fullname = 'Frodo Baggins'
print(emp1.fullname)

del emp1.fullname
print(emp1.fullname)
```

Output:

```
Sam Gamgee
Frodo Baggins
None None
```

- `@property` converts a method to something like an attribute.
- `@methodname.setter` enable to write set value to an attribute/method defined by `@property`.
- `@fullname.deleter` Clears the values when we delete an attribute/method using `del`

Getting and setting attributes.

- `dir(MyClass)`: List all the attributes in a class.
- `f = getattr(obj, 'foo')`.
- `setattr(obj, 'foo', bar)`.