Python Files and modules

Petr Svarny, 2020

Module, package, library

- There is no strict classification
- Module is file or folder containing code
  - E.g. text file with .py ending
- Package is usually set of several modules
- Library is general name for package, used in other languages also

Module import

- Import whole module using ***import*** *module*, call function as module.function()

```
>>> import os
>>> os.getcwd()
'/home/me'
```

- Import only one function **from** module **import** function

```
>>> from os import getcwd
>>> getcwd()
'/home/me'
```

Module import

- Using abbreviation ***import*** *module* ***as*** *mod*, call function as mod.function

```
>>> import pandas as pd
>>> pd.read_table('apple.txt')
```

- Load module from package as **import** mod.submodule as mod

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y)
```

Module import error

```
>>> import Bio

-----------------------------------------------------------------

ImportError    Traceback (most recent call last)

<ipython-input-1-a7440e1156be>  in <module>()

----> 1 import  Bio
```

Import module

```
>>> from datetime import datetime

>>> now_time = datetime.now()
```

Modules

- Standard library
  - Already installed
  - E.g.: `math, os, sys, random`
  - More info: https://docs.python.org/3.6/library/
- Other modules
  - Install using pip
    - `sudo pip3 install module_name`
    - `sudo pip install module_name`
    - E.g.: pandas, numpy, matplotlib, plotly

Create your own module

Note: module must be in the same directory, or in directory above

hello.py

```
def print_hello():
    print('Hello!')
```

main_program.py

```
import hello
hello.print_hello()
```

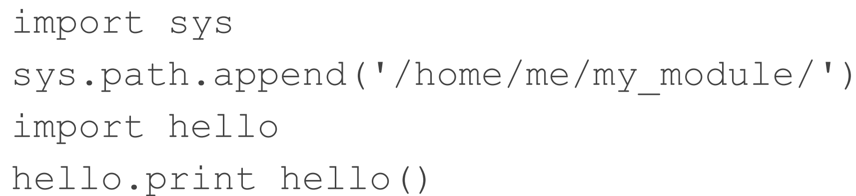
Create your own module

If module is in

/home/me/my_modules/hello.py

```
def print_hello():
    print('Hello!')
```

/home/me/my_scripts/main_program.py



```
import sys
sys.path.append('/home/me/my_module/')
import hello
hello.print_hello()
```

Exercise

- Create python file that will contain function `divide_two_numbers`
- Import this function to a different Python file, main.py, or Jupyter Notebook
- Call function in the second file main.py or in your Jupyter Notebook, e.g. `divide_two_numbers(3,5)`

Run python script from command line

Create `my_script.py` (e.g. in text editor).

```
$ cat my_script.py

print("Hello world!")
```

Run script using `python3`

```
$ python3 my_script.py

Hello world!
```

Run python script from command line

- Use **sys.argv** from sys package
- sys.argv is the list of command-line arguments,
  the program name is first argument, i.e. sys.argv[0]

```
import sys
def sum_num(a,b):
        return a+b

print(sum_num(int(sys.argv[1]),int(sys.argv[2])))

$ python my_script.py 3 2
5
```

Argparse

- Library for parsing arguments from the command-line
- Allows easy help integration
- Use of positional or **optional** arguments
- See the documentation

Argparse import and start

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

```
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Argparse positional arguments

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

```
$ python3 prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python3 prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py foo
foo
```

Argparse positional arguments with help

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

```
$ python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

optional arguments:
  -h, --help  show this help message and exit
```

Argparse positional arguments with type

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

```
$ python3 prog.py 4
16
$ python3 prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

Argparse optional arguments

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

```
$ python3 prog.py --verbosity 1
verbosity turned on
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python3 prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Argparse optional arguments with actions

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

```
$ python3 prog.py --verbose
verbosity turned on
$ python3 prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python3 prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Argparse optional arguments with short options

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

Argparse combining arguments

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print("the square of {} equals {}".format(args.square, answer))
else:
    print(answer)
```

```
$ python3 prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python3 prog.py 4
16
$ python3 prog.py 4 --verbose
the square of 4 equals 16
$ python3 prog.py --verbose 4
the square of 4 equals 16
```

Argparse combining arguments and defaults

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

Exercise

Write a small Python script `count_letters.py` using argparse that:

- Has a positional argument of the string in which letters are supposed to be counted.
- Has two optional arguments:
  - v: to count only vowels
  - c: count only consonants

The script prints out a list of letters in alphabetical order with the number of occurrences:

a 2

b 5

Script structure

Often the script contains a function called “main” just to be clear what is the purpose of the script. This is, however, not necessary.

On the other hand, the following block can be used to make sure the script’s code runs only in case the script is directly called:

```
if __name__ == ’__main__’:
  <body that is meant to run>
```

Exercise

- Write script
- Input values: two strings as arguments from command line
- Script will print number of occurrences of substring in string
- Example:

```
$python count_occurrence.py ab abcdabcc
String ab occurred 2 times in string abcdabcc.
```

Exercise

- Write script
- Input values: two strings (word and letter) as arguments from command line
- Script will print word without specified letter
- Example:

```
$python extract_letter.py python o
pythn

$python extract_letter.py python l
python
```

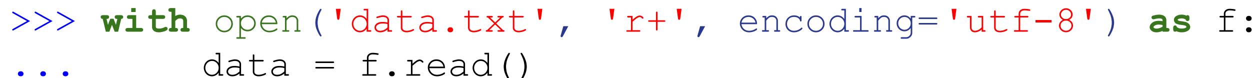
Working with files - open

- The same process as we work with the file
    - Open file -> Action (read, write, edit) -> Close file
- ***open*** function will create file object
    - `open(filename including path, mode)`
- ***open*** has several modes (can be combined)
    - **'r'- file is opened for reading, error if file does not exist**
    - 'r+' - file is opened for reading and writing, error if file does not exist
    - **'w' - file is opened for writing, existing file will have zero length, if file does not exist, new file will be created**
    - 'a' - file is opened for appending at the end of file, if file does not exist, new file will be created
    - 'b' - file will be open in binary mode, e.g. photos, movies.

Working with files - close

- To close file use ***close*** method
    - `file.close()`
- **Do not forget to close file, lead to file truncation!**
- Solution: use ***with*** statement
    - File will close automatically after with statement

```
>>> f = open('data.txt', 'r+', encoding='utf-8')
>>> data = f.read()
>>> f.close()
```



Working with files - write

- Use ***write*** method

```
>>> with open('hello.txt', 'w', encoding='utf-8') as f:
... print(type(f))
... f.write('Hello world!')

>>> fruits = ['apple', 'pear', 'apricot', 'banana', 'kiwi']

>>> with open('fruits.txt', 'w', encoding='utf-8') as f:
... for fruit in fruits:
...     f.write(fruit + '\n')
```

Working with files - read

- Several functions are available
  - ***read*** read the whole file as one string
  - ***readlines*** read the whole file as list of lists
  - ***readline*** read one line and return string

```
>>> with open('fruits.txt', 'r', encoding='utf-8') as f:
...     fruit_data = f.read()
>>> fruit_data
'apple\npear\napricot\nbanana\nkiwi\n'
```

Working with files - read

```
>>> with open('fruit.txt', 'r', encoding='utf-8') as f:
...     fruit_data = f.readlines()
>>> fruit_data
['apples\n', 'apricots\n', 'peaches\n', 'bananas\n']

>>> with open('fruit.txt', 'r', encoding='utf-8') as f:
...     fruit_data = f.readline()
>>> fruit_data
'apples\n'
```

Working with files - read

```
>>> with open('fruits.txt', 'r', encoding='utf-8') as f:
...     fruit_data = f.read().splitlines()

>>> fruit_data
['apples', 'apricots', 'peaches', 'bananas']
```

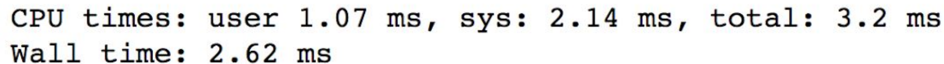
Offtopic: timing functions in Jupyter Notebook

- Use Jupyter notebook magic function %%timeit or %%time
- More information here

In [66]:
```
%%timeit
with open('fruits.txt', 'r', encoding='utf-8') as f:
    fruit_data = f.read().splitlines()
```

```
60.9 µs ± 15.9 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [69]:
```
%%time
with open('fruits.txt', 'r', encoding='utf-8') as f:
    fruit_data = f.read().splitlines()
```



Exercise

- Create list of things you would like to take on the empty island
- Write this list to the tab-delimited file, so that each element is on a different line and lines are numbered
- Example

```
1	casserole
2	book
3	knife
4	water bottle
5	fishing rod
```

- Hint: you can use `print` with additional parameters