Moduł – to skrypt w Pythonie, jeden plik

Using modules

PEP 302 - New Import Hooks

INTRODUCTION

Why are we using modules?
(a) To reuse code. #ponowne użycie kodu, podobnie jak funkcje ale takie w innych plikach
(b) To keep the core language small.
(c) To make a safe namespace.
(d) For shared services and data.

Moduły się tworzy jako pliki

A Python program consists of the top-level module and zero or more additional modules.

'__main__' is the name of the scope in which top-level code executes. A module's __name__ is set equal to '__main__' when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own __name__.

```
File name: 'starwars.py' #nazwa pliku na dysku
Module name: 'starwars' #nazwa modułu zaimportowanego
__name__ == '__main__' if the module is running in the main scope ##zmienna #name ma nazwę main, jeśli jesy jedyna
__name__ == 'starwars' if the module is imported
```

---

import statements

https://docs.python.org/3/reference/import.html

https://docs.python.org/3/library/importlib.html

INTRODUCTION

Python code in one module gains access to the code in another module by the process of importing it.

The 'import' statement combines two operations:
(a) searching for the named module [finders],
(b) binding the results of that search to a name in the local scope [loaders; module execution].

When a module is first imported, Python searches for the module and if found, it creates a module object, initializing it. If the named module cannot be found, a 'ImportError' (Py2) or 'ModuleNotFoundError' (Py3.6+) is raised.

The Python interpreter searches for modules in the following sequences:
(a) the current directory,
(b) directories in the shell variable PYTHONPATH.
(c) default locations (installation-dependent).

The module search path is stored in the system module 'sys' as the 'sys.path' variable.

Pliki modułów poszukkuje się w katalogach sys.path

```
import sys

print ( sys.modules.keys() )    # names of imported modules
print ( sys.path )             # the module search path
```

```
# Using 'import' statements.

import module1   # recommended
import module2, module3

print ( module1.name1 )
other_name = module1   # modules are objects
print ( other_name.name1 )
module1.name1 = new_value     # be careful!
module1.name2()       # a function call
print ( module1.__doc__ )   # docstring
help(module1)   # NAME, FILE, FUNCTIONS, DATA
dir(module1)               # namespace
module1.__dict__.keys()   # namespace
```

```
# Using 'from...import' statements.
# We can import specific names from a module into the current namespace.
#nie jest tworzona przestrzeń nazw
#obiekt „module” nie istnieje

from module1 import name1, name2

# Equivalent statements:
# import module1
# name1 = module1.name1
# name2 = module1.name2
# del module1

from module2 import *   # importing all names, possible problems

print ( "{} {}".format(name1, name2()) )
```

```
import module1 as module2

# Equivalent statements:
# import module1
# module2 = module1
# del module1

import numpy as np
import Tkinter as tk   # Py2
import tkinter as tk   # Py3
import pygame as pg
```

```
from module3 import name1 as name2
```

```
# The reload() function imports a previously imported module again.

#reload(module1)   # Py2

from importlib import reload   # Py3.4+
reload(module1)
```

Creating modules

INTRODUCTION

```
#!/usr/bin/python   # optional

module_docstring   # optional

# imports
import sys

# constants

# exception classes

# interface functions

# classes

# internal functions and classes

def tester():   # a function with tests, optional
    statements

if __name__ == "__main__":   # run tests if we are in the main module
    status = tester()
    sys.exit(status)
```

Packages

https://docs.python.org/3/reference/import.html

PEP 420 - Implicit Namespace Packages [a regular package (with an __init__.py) vs a namespace package]

#pogrupowanie modułów

INTRODUCTION

Python has a concept of 'packages' to help organize modules and provide a naming hierarchy. You can think of packages as the directories on a file system and modules as files within directories.

It’s important to keep in mind that all packages are modules, but not all modules are packages [a package contains a '__path__' attribute].

Python defines two types of packages, 'regular packages' and 'namespace packages'.

REGULAR PACKAGES

Regular packages are traditional packages typically implemented as a directory containing an '__init__.py' file.

When a regular package is imported, this '__init__.py' file is implicitly executed, and the objects it defines are bound to names in the package’s namespace. The '__init__.py' file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

```
# Files in the current directory.

script.py       # the main module
dir1/__init__.py       # required trzeba mieć taki plik żeby katalog był pakietem

dir1/module1.py
dir1/dir2/__init__.py   # required
dir1/dir2/module2.py
```

```
# Possible statements in script.py

import dir1
import dir1.module1
import dir1.dir2.module2

from dir1.module1 import name1
from dir1.dir2.module2 import *
```

NAMESPACE PACKAGES

A namespace package is a composite of various 'portions', where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system. Portions may also be found in zip files, on the network, or anywhere else that Python searches during import. Namespace packages may or may not correspond directly to objects on the file system; they may be virtual modules that have no concrete representation.

SYMPY

https://www.sympy.org/

https://github.com/sympy/sympy
[GitHub account: sympy, GitHub repo: sympy]

SymPy is the Python Library for Symbolic Mathematics.

```
sympy/          # main directory
sympy/__init__.py   # not empty

sympy/core/
sympy/core/__init__.py   # not empty
sympy/core/basic.py

sympy/core/tests/
sympy/core/tests/__init__.py   # empty
sympy/core/tests/test_basic.py

sympy/combinatorics/
sympy/combinatorics/__init__.py   # not empty
sympy/combinatorics/permutations.py
# from sympy.core.basic import Atom

sympy/combinatorics/tests/
sympy/combinatorics/tests/__init__.py   # empty
sympy/combinatorics/tests/test_permutations.py
# from sympy.combinatorics.permutations import Permutation
```

Using os #operating system

https://docs.python.org/3/library/os.html

https://docs.python.org/3/library/os.path.html

INTRODUCTION

OS module provides various functions to interact with the operating system. Programs that import and use 'os' stand a better chance of being portable between different platforms. Programs should leave all pathname manipulation to 'os.path' (e.g., split and join).

```
import os

# current working directory
current = os.getcwd()   # '/home/andrzej' in Linux Debian

# Using join().
# parent directory
print ( os.pardir )   # ".." in UNIX
parent = os.path.join(current, os.pardir)   # '/home/andrzej/..'
# Join two or more pathname components, inserting os.sep as needed.

print ( current + os.sep + os.pardir )   # the same
print ( os.path.abspath(parent) )   # '/home', the absolute path
print ( os.path.relpath(parent) )   # '..' the relative path
```

```
# Using split().

path = '/home/andrzej/Pobrane'
print ( os.path.exists(path) )   # True
print ( os.path.isabs(path) )   # True, it is the absolute path
print ( os.path.isfile(path) )   # False
print ( os.path.isdir(path) )   # True
print ( os.path.getsize(path) )   # the number of bytes

print ( os.path.split(path) )   # ('/home/andrzej', 'Pobrane')
# Split a pathname.  Returns tuple "(head, tail)" where "tail" is
# everything after the final slash.  Either part may be empty.

print ( os.path.basename(path) )   # 'Pobrane'
print ( os.path.dirname(path) )   # '/home/andrzej'
```

```
# Create a directory.

dirname = "MyWork"
print ( os.path.isdir(dirname) )   # False
os.mkdir(dirname)
# os.rename(old_name, new_name)
print ( os.path.isdir(dirname) )   # True
print ( os.path.abspath(dirname) )   # '/home/andrzej/MyWork'
os.rmdir(dirname)   # a directory have to be empty

# Create directory and all intermediate directories if don't exists.
dirname = "A/B/C"
if not os.path.exists(dirname):
    os.makedirs(dirname)
```

EXAMPLES

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/',
# it could delete all your disk files.

import os

for root, dirs, files in os.walk(top, topdown=False):   # walking bottom-up
    # 'root' is a string, the path to the directory
    # 'dirs' is a list of the names of the subdirectories in 'root'
```

```
    # 'files' is a list of the names of the non-directory files in 'root'
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

```
# Display the number of bytes taken by non-directory files in each
directory
# under the starting directory ('top'). Skip any CVS subdirectory.

import os

for root, dirs, files in os.walk(top):   # walking top-down (default)
    print(root, "consumes", end=" ")
    print(sum(os.path.getsize(os.path.join(root, name)) for name in files),
end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

```
# For a given directory find the number of PDF files in the directory tree
# (".pdf" extensions).

import os

n_pdf = 0   # the number of PDF files
for root, dirs, files in os.walk(top):   # walking top-down (default)
    for name in files:
        #if name[-4:] == ".pdf":
        #if name.endswith(".pdf"):
        if name.lower().endswith(".pdf"):   # for *.PDF files
            n_pdf += 1

print("The number of PDF files in {} directory is {}".format(top, n_pdf))
```

Using sys

https://docs.python.org/3/library/sys.html

INTRODUCTION

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

```
import sys

# Dynamic objects (selected).

sys.argv   # command line arguments; argv[0] is the script pathname if
known
sys.path   # module search path (a list of strings)
# sys.path[0] is the script directory, else ''
```

```
sys.modules   # dictionary of loaded modules

sys.stdin   # standard input file object
sys.stdout   # standard output file object
sys.stderr   # standard error object; used for error messages
```

```
# Static objects (selected).

sys.executable   # absolute path of the executable binary of the Python
interpreter
sys.maxint   # the largest supported integer
sys.version   # the version of this interpreter as a string
sys.platform   # platform identifier
sys.prefix   # prefix used to find the Python library
```

```
# Functions (selected).

sys.exit()   #  exit the interpreter by raising SystemExit
# sys.exit(0) is a success, sys.exit(1) is a failure
sys.getrefcount()   # return the reference count for an object (plus one :-
)
sys.getrecursionlimit()   # return the max recursion depth for the
interpreter
sys.getsizeof()   # return the size of an object in bytes
sys.setrecursionlimit()   # set the max recursion depth for the interpreter
```

EXAMPLES

```
#!/usr/bin/python
# script.py

import sys

print(sys.argv)
```

```
$ ./script.py 12 34
['./script.py', '12', '34']

$ python3 script.py 12 34
['script.py', '12', '34']
```

---

to jest kopia stworzona na potrzeby skryptu fid_pdf_size