Școala
informală
de IT

Python Development

Conditional programming, loops & functions

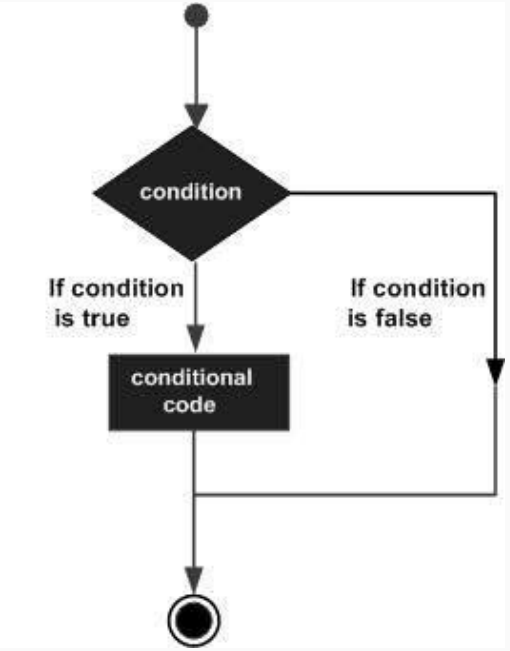
Cuprins

Programare condițională

Conditional programming, loops & functions

Programare condițională

- Programarea condițională reprezintă anticiparea condițiilor care pot avea loc în timpul executării unui program și specificarea acțiunilor care trebuie luate.
- Structurile decizionale evaluează mai multe expresii care au o valoare booleană: **True** sau **False**. În urma condiției trebuie executată o instrucțiune sau un set de instrucțiuni.



- Python atribuie:
  - valoarea **True** oricărei valori non-zero și non-null.
  - valoarea **False** oricărei valori zero sau null.
- Programarea condițională se face folosind instrucțiunile **if...elif...else**.
- Pentru luarea unei decizii poate fi folosită doar instrucțiunea if, ramura else fiind opțională.
- Pentru înșiruirea ramurilor decizionale se folosește **if...elif** - elif este keyword-ul Python pentru *else if*-ul din alte limbaje.

```
my_var = 5
if my_var < 6:
    print("Set Instructiuni #1")
```

```
my_var = 5
if my_var < 6:
    print("Set Instructiuni #1")
elif my_var < 10:
    print("Set Instructiuni #2")
```

```
my_var = 5
if my_var < 6:
    print("Set Instructiuni #1")
elif my_var < 10:
    print("Set Instructiuni #2")
elif my_var < 25:
    print("Set Instructiuni #3")
else:
    print("Set Instructiuni #4")
```

Structuri repetitive

Conditional programming, loops & functions

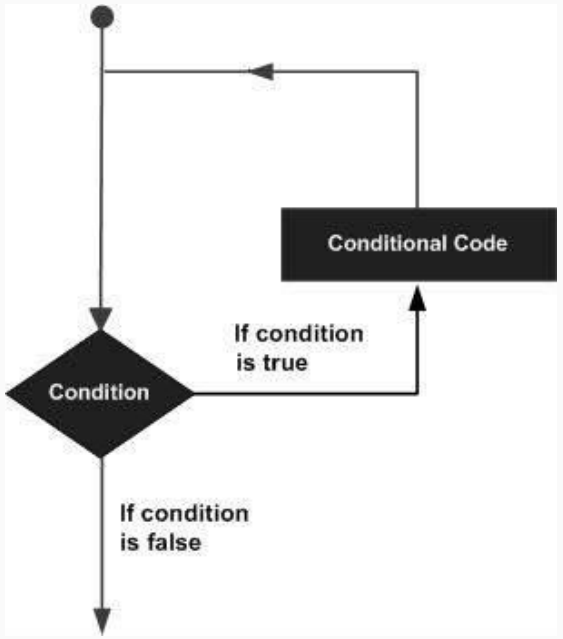
Structuri repetitive

- În mod normal instrucțiunile sunt executate secvențial. Codul este rulat linie cu linie.
- În practică, există nevoia ca un grup de instrucțiuni să fie executat de mai multe ori.
- O structură repetitivă ne permite să executăm o instrucțiune sau un grup de instrucțiuni de mai multe ori.
- Limbajul Python oferă următoarele două astfel de structuri:
  - while - repetă un set de instrucțiuni atât timp cât este îndeplinită o condiție.

```
while True:
    print("Set Instructiuni")
```

    - mai întâi este verificată condiția, iar apoi se execută codul.
    - este o structură repetitivă cu număr necunoscut de pași. Codul va fi executat cât timp condiția este îndeplinită - nu vom ști de câte ori.
  - for - repetă un set de instrucțiuni de un număr cunoscut și finit de pași.

```
for i in range(10):
    print(f'Set Instructiuni [{i + 1}]')
```



Structuri repetitive

- Există trei instrucțiuni cu ajutorul cărora putem controla modul de executare al structurilor repetitive.
- Rolul acestora este de a schimba modul secvențial de executare al instrucțiunilor.
  - break - oprește execuția buclei si transferă controlul către următoarea instrucțiune din afara buclei.

```
while True:
    random_choice = random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    if random_choice % 3 == 0:
        break
    print(f'random_choice = {random_choice}')
```

  - continue - oprește executarea restului de cod, dar transferă controlul următoarei iterații.

```
for i in range(10):
    if i % 2 != 0:
        continue
    print(f'Numar par: {i}')
```

  - pass - este o instrucțiune ce are rol de placeholder. Nu are absolut nici o acțiune, doar substituie conținutul unui bloc pentru a permite scrierea acestuia, dar necompletarea lui cu instrucțiuni.

```
if True:
    pass
```

Funcții

Conditional programming, loops & functions

Funcții

- O funcție reprezintă un bloc organizat de cod ce poate fi refolosit și are rolul de a realiza o singură acțiune.
- Funcțiile oferă o modularitate mai bună a aplicației și un mare avantaj în refolosirea codului.
- Python conține multe funcții predefinite, dar fiecare utilizator își poate crea propriile funcții. Câteva exemple de funcții predefinite:
  - **print()** - este funcția cu care afișăm un mesaj în consolă.
  - **format()** - este funcția cu care formatăm un șir de caractere.
  - **input()** - este funcția cu care citim date introduse de la tastatură.

Funcții

- Prin semnătura unei funcții înțelegem linia de cod care definește funcția respectivă. Cea mai abstractă semnătură a unei funcții în Python este următoarea:

```
def my_function(*args, **kwargs):
    pass
```

- În exemplul de mai sus observăm declarată o funcție care nu face nimic - blocul de instrucțiuni conține doar instrucțiunea **pass**. Totuși din această formă complet abstractă putem trage următoarele concluzii:
  - pentru declararea unei funcții trebuie să folosim keyword-ul **def**. Acesta va fi primul cuvânt din semnătura unei funcții.
  - următorul cuvânt reprezintă numele funcției. Acesta trebuie să respecte aceleași reguli ca numele unei variable:
    i. vom folosi notarea snake_case.
    ii. nu poate fi același cu un keyword Python.
    iii. nu poate începe cu o cifră
    iv. poate conține orice înșiruire din a-z, A-Z, 0-9 și caracterul underscore _.
  - următoarea parte din semnătura funcției o reprezintă lista de parametrii. Aceasta poate fi goală, **()**, dacă funcția nu conține nici un parametru, altfel parametrii trebuie înșiruiți între paranteze **()**.
  - reprezentând un bloc de cod, semnătura oricărei funcții se termină cu caracterul :

Funcții

- O funcție poate întoarce un rezultat, dar acest lucru nu este obligatoriu.
- Să luăm exemplul unei funcții care returnează suma a două numere.
  - suma poate fi calculată de o funcție, iar rezultatul obținut poate fi returnat. Pentru a returna o valoare vom folosi keyword-ul **return**.

```
def get_sum(a, b):
    return a + b


my_sum = get_sum(2, 5)
print(my_sum)  # va afisa 7
```

  - suma poate fi calculată folosind o variabilă globală, astfel funcția noastră doar va calcula suma celor numere, dar nu va returna rezulatul. Rezultatul va fi notat în variabila globală **my_sum**.

```
my_sum = 0


def get_sum(a, b):
    global my_sum
    my_sum = a + b


get_sum(2, 5)
print(my_sum)  # va afisa 7
```

- Pentru a apela o funcție, așa cum se poate observa în exemplele anterioare, vom folosi numele funcției și vom transmite parametrii.

Funcții

- Un aspect foarte important legat de funcții o reprezintă lista de parametrii.
- În primul rând trebuie să știm că parametrii unei funcții sunt trimiși prin referință. Asta înseamnă că orice modificare a parametrului în cadrul funcției se va reflecta și în-afara acesteia.

```
def my_function(list_param):
    list_param.append(4)
    # consider doing something else here...


my_list = [1, 2, 3]
my_function(my_list)
print(my_list)  # va afisa [1, 2, 3, 4]
```

- În exemplul de mai sus, variabila my_list a fost trimisă funcției my_function, iar în interiorul acesteia valoarea ei a fost modificată. Din câte puteți observa s-a modificat și valoarea parametrului my_list după apelarea funcției.
- A se nota că acest lucru nu este valabil dacă facem o reasignare a parametrului respectiv datorită modului de lucru al Python Management Memory.

```
def my_function(list_param):
    list_param = list_param.copy()
    list_param.append(4)
    # consider doing something else here...


my_list = [1, 2, 3]
my_function(my_list)
print(my_list)  # va afisa [1, 2, 3]
```

Funcții

- Un alt aspect foarte important ce ține de parametrii unei funcții o reprezintă tipul acestora.
- Din punct de vedere al modului în care sunt declarați și transmiși aceștia pot fi împărțiți astfel:
  - **poziționali (required)**. Aceștia apar primii în lista de parametrii, iar trimiterea lor este obligatorie.

```
def my_function(param_1, param_2):
    pass


# acest apel va rezulta intr-o eroare:
# TypeError: my_function() missing 1 required positional argument: 'param_2'
my_function(2)
```

    - ordinea în care sunt trimiși trebuie respectată.

```
def my_function(param_1, param_2):
    print(param_1)  # va afisa 2
    print(param_2)  # va afisa 5


my_function(2, 5)
```

```
def my_function(param_1, param_2):
    print(param_1)  # va afisa 5
    print(param_2)  # va afisa 2


my_function(5, 2)
```

Funcții

- **cheie-valoare (key=value)**. Aceștia apar în listă după parametrii poziționali și sunt setați sub forma **cheie=valoare**. Astfel prezența lor în apelul funcției nu mai este obligatorie. Dacă aceștia lipsesc din apelul funcției valoarea default va fi folosită. Ordinea lor nu este importantă, deoarece sunt specificați prin nume, dar trebuie să succeadă parametrii poziționali.

```
def my_function(param_1, param_2, param_3=0):
    print(param_1)  # va afisa 2
    print(param_2)  # va afisa 5
    print(param_3)  # va afisa -3


my_function(2, 5, -3)
```

```
def my_function(param_1, param_2, param_3=0):
    print(param_1)  # va afisa 2
    print(param_2)  # va afisa 5
    print(param_3)  # va afisa 0


my_function(2, 5)
```

Funcții

- Dacă vă aduceți aminte, la începutul acestui capitol am vorbit de forma cea mai abstractă a unei funcții.

```
def my_function(*args, **kwargs):
    pass
```

- Din câte puteți observa în exemplul de mai sus, parametrii specificați nu prea respectă tiparul folosit în exemplificările anterioare
- Asta pentru că în Python putem folosi variable-length arguments. Cu alte cuvinte, există acești parametri speciali (precedați cu *, respectiv ** - numele poate fi altul)
    - primul parametru, precedat cu *, are rolul de a prelua toți parametrii poziționali nedeclarați în semnătura funcției, dar transmiși în momentul apelării acesteia. Ei se vor regăsi într-o listă în ordinea în care au fost transmiși.

```
def my_function(param_1, param_2, *args):
    print(param_1)  # va afisa 2
    print(param_2)  # va afisa 5
    print(args)  # va afisa [-3, 6, 'abc']


my_function(2, 5, -3, 6, 'abc')
```

    - al doilea parametru, precedat cu **, are rolul de a prelua toți parametrii cheie:valoare nedeclarați în semnătura funcției, dar transmiși în momentul apelării acesteia. Ei se vor regăsi într-un dicționar.

```
def my_function(param_1, param_2, **kwargs):
    print(param_1)  # va afisa 2
    print(param_2)  # va afisa 5
    print(kwargs)  # va afisa {"a: 2, "b": -3}


my_function(2, 5, a=2, b=-3)
```

Funcții

- O caracteristică specifică funcțiilor o reprezintă recursivitatea.
- O funcție este recursivă dacă se apelează singură.
- Exemplul clasic al recursivității este următorul:

```
def get_sum(n):
    if n == 0:
        return 0

    return n + get_sum(n - 1)


# 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28
print(get_sum(7))  # va afisa 28
```

- Funcția anterioară calculează suma tuturor numerelor cuprinse în intervalul **[0, n]**.
- O funcție recursivă trebuie să îndeplinească două caracteristici:
  - să se auto-apeleze
  - să conțină o condiție pentru oprirea recursivității.

Tratarea excepțiilor

Conditional programming, loops & functions

Tratarea excepțiilor

- O excepție este un eveniment care are loc în timpul executării unui program, în urma acestuia fiind întreruptă executarea programului.
- Atunci când interpretorul întâlnește o situație pe care nu știe să o gestioneze, acesta aruncă o excepție.
- Excepția este un obiect care reprezintă o eroare.
- Când un program scris în Python aruncă o eroare, aceasta trebuie tratată imediat, altfel programul se termină instant.
- Tratarea excepțiilor se face atunci când codul scris poate întâmpina o eroare, această abordare reprezentând o măsură de precauție a developerului.

Tratarea excepțiilor

- Tratarea excepțiilor se face folosind blocul **try...except**.

```
my_var = input("int number: ")
try:
    my_int = int(my_var)  # Will raise a ValueError if from keyboard is not passed an int.
    print(not_defined_variable)  # Will raise a NameError since the variable doesn't exist.
except ValueError as e:
    print('Do something with ValueError exception.', e)
except NameError as e:
    print('Do something with NameError exception.', e)
else:
    print('Reaches here if there is no exception.')
finally:
    print('Reaches here no matter if there is an exception or not.')
```

  - ramura **try** este folosită pentru a rula codul care ne interesează. Acest cod poate fi problematic și poate arunca excepții.
  - ramura **except** este folosită pentru a prinde excepția și a o trata. Dacă nu se dorește tratarea acesteia, blocul except poate conține doar instrucțiunea **pass**, dar prinderea excepției este obligatorie.
  - ramura **else** este folosită pentru executarea unor instrucțiuni când codul din ramura try a funcționat fără probleme. Nu este obligatorie prezența acesteia.
  - ramura **finally** se folosește pentru rularea unor instrucțiuni indiferent dacă a fost aruncată sau nu o excepție. Nu este obligatorie prezența acesteia.

Namespaces

Conditional programming, loops & functions

Namespaces

- Un namespace este o colecție de link-uri simbolice împreună cu informația aferentă fiecărui obiect referențiat de acesta.
- Vă puteți gândi la un namespace ca la un dicționar în care cheia este numele obiectului, iar valoarea o reprezintă obiectul. Fiecare pereche cheie:valoare mapează un nume cu obiectul aferent.
- Într-un program dezvoltat în Python există patru tipuri de namespace-uri:
  - built-in
  - global
  - enclosing
  - local
- Fiecare dintre acestea au propriul ciclu de viață. Când este executat un program dezvoltat în Python, aceste namespace-uri sunt create când este nevoie de ele și șterse când nu mai sunt necesare.
- În general, mai multe namespace-uri vor exista în orice moment al rulării programului.

Namespaces - built-in

- Built-in namespace-ul conține numele tuturor obiectelor Python predefinite. Acesta este disponibil în orice moment al rulării unui program dezvoltat în Python. O listă completă cu toate aceste obiecte poate fi obținută folosind comanda *dir(__builtins__)*:

```
Python 3.8.5 (default, Jul 20 2020, 19:48:14)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError'
, 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotF
oundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', '
KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'Overfl
owError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteratio
n', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncod
eError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '
__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex'
, 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help
', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', '
print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

- Dacă veți căuta cu atenție prin această listă o să întâlniți multe nume deja folosite în aceste cursuri: **print, int, len, complex, ValueError, NameError**

Namespaces - global

- Namespace-ul global conține orice obiect definit la nivelul programului principal (*main*).
- Acest namespace este creat în momentul în care programul principal este rulat și rămâne activ până în momentul în care programul se termină.
- Un namespace global nu va fi unic. Interpretorul creează un global namespace pentru fiecare modul importat de programul nostru (vom vorbi despre asta în capitolul următor).

Namespaces - local

- De fiecare dată când o funcție este creată, se creează un namespace local funcției respective.
- Un astfel de namespace local este creat în momentul în care funcția este apelată și sters odată cu terminarea execuție sale.

```
def my_function():
    msg = 'Hello, World!'
    print(msg)  # va afisa "Hello, World!"


my_function()
print(msg)  # variabila nu exista
```

- În exemplul anterior, în blocul funcției **my_function** este declarată variabila **msg**. Dacă veți rula codul anterior o să primiți o eroare pentru că variabila msg nu este definită în afara funcției. În aceast caz, variabila **msg** se află în namespace-ul local al funcției **my_function.**
- Pentru a folosi o variabilă globală va trebui să folosim keyword-ul **global** în interiorul funcției.

```
def my_function():
    global msg
    msg = 'Hello, World!'
    print(msg)  # va afisa "Hello, World!"


my_function()
print(msg)  # va afisa "Hello, World!"
```

Namespaces - enclosing

```
def my_function():
    def my_second_function():
        # va afisa "my_second_function: Hello, World!"
        print(f'my_second_function: {msg}')

    msg = 'Hello, World!'

    my_second_function()

    # va afisa "my_function: Hello, World!"
    print(f'my_function: {msg}')


my_function()
```

- În exemplul alăturat aveți definite două funcții imbricate (nested):
  - funcția my_function este **enclosing** function
  - funcția my_second_function este **enclosed** function.
- Din câte puteți observa funcția **my_second_function** folosește variabila **msg** din namespace-ul funcției **my_function**.
- Namespace-ul funcției **my_function** este enclosing namespace-ul funcției **my_second_function**.

Namespaces

- Având în vedere aceste informații, vom ști că putem defini mai multe variabile folosindu-ne de același nume.
- Luând ca exemplu variabila x, interpretorul Python va decide către cine să pointeze în momentul folosirii acestei variabile.
- Această variabile poate exista în mai multe namespace-uri:
  - global
  - enclosing
  - local
- Ordinea în care interpretorul Python va căuta referința x va fi inversă față de exemplul anterior:
  - local
  - enclosing
  - global

Module și pachete

Conditional programming, loops & functions

Module și pachete

- Un modul este un fișier Python (cu extensia .py) ce are rolul grupării codului pentru o organizare mai bună a acestuia.
- Un modul poate conține un număr infinit atât de variabile, funcții și clase cât și cod executabil.
- Fiecărui modul îi este atribuită o variabilă numită **__name__**. Roul acesteia este de a identifica modulul respectiv:
  - numele unui modul este dat de numele fișierului .py în cazul în care acesta este importat.
  - numele unui modul este **__main__** dacă fișierul este rulat individual (ca script).
- Spre deosebire de alte limbaje de programare, Python nu conține un entrypoint (o funcție main) de unde începe rularea programului. Într-un program Python orice modul al cărui **__name__** este **__main__** poate reprezenta un punct de start.

```
if __name__ == 'main':
    print('This is the main functionality.')
```

Module și pachete

- Folosiți-vă de module pentru a vă organiza codul și pentru a nu încărca scriptul cu toată funcționalitatea.
- Pentru a exemplifica lucrul cu module vom considera următorul modul ca exemplu:

```
my_var = 5



def my_func():
    return f'I am a function from {__name__}.'
```

- Obiectele dintr-un modul pot fi importate folosind instrucțiunea **import**. Aceasta dispune de mai multe variante:
    - importarea completă a modulului (există posibilitatea etichetării acestuia folosind keyword-ul **as**).

      Ambele exemple de mai jos vor produce același rezultat.

```
import my_module

if __name__ == '__main__':
    print(f'Use another module variable: {my_module.my_var}')
    print(f'Call another module function: {my_module.my_func()}')
```

```
import my_module as tagged_module

if __name__ == '__main__':
    print(f'Use another module variable: {tagged_module.my_var}')
    print(f'Call another module function: {tagged_module.my_func()}')
```

Module și pachete

- importarea tuturor obiectelor dintr-un modul (implică riscul rescrierii unor obiecte din modulul curent sau din alte module importate):

```
from my_module import *

if __name__ == '__main__':
    print(f'Use another module variable: {my_var}')
    print(f'Call another module function: {my_func()}')
```

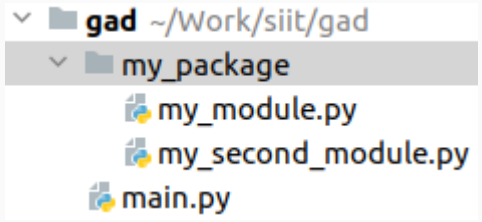
- importarea individuală dintr-un modul (fiecare obiect poate fi taguit folosind keyword-ul **as**):

```
from my_module import my_var, my_func

if __name__ == '__main__':
    print(f'Use another module variable: {my_var}')
    print(f'Call another module function: {my_func()}')
```

```
from my_module import my_var as tagged_var, my_func

if __name__ == '__main__':
    print(f'Use another module variable: {tagged_var}')
    print(f'Call another module function: {my_func()}')
```

Module și pachete

- Un pachet Python reprezintă un director care conține alte pachete și module Python. Rolul acestora este de a organiza codul și a menține o structură cât mai logică a proiectului.
- Până în versiunea Python 3.2, pentru a folosi un pachet era necesară prezența unui fișier special ce avea numele **__init__.py**. Acest fișier putea fi gol, dar prezența lui era obligatorie.
- Începând cu versiunea Python 3.3, nu mai este o necesitate prezența acestui fișier.
- În exemplul anterior puteți vedea cum arată un pachet care conține două module:



- Pentru folosirea obiectelor din cele două module putem folosi următoarele linii de import:

```
from my_package.my_module import my_var as tagged_var, my_func
from my_package.my_second_module import my_second_var
```

Module și pachete

- Deși prezența fișierului __init__.py nu mai este obligatorie începând cu Python 3.3, de multe ori acest fișier este necesar.
- Necesitatea acestuia se explică prin modul în care codul din acest fișier este rulat. Codul din acest fișier este rulat atunci când este importat un obiect dintr-un modul ce ține de pachetul respectiv.
- Astfel apar două motive pentru care avem nevoie de acest fișier:
  - rularea unui bloc de cod. Ex: diferite configurări.
  - expunerea obiectelor disponibile în module astfel încât atunci când se va face un import să nu mai fie nevoie să parcurgem toată calea către modulul dorit.

    → expunem tot ce ne interesează din module prin importarea obiectelor în fișierul de __init__.py al pachetului.

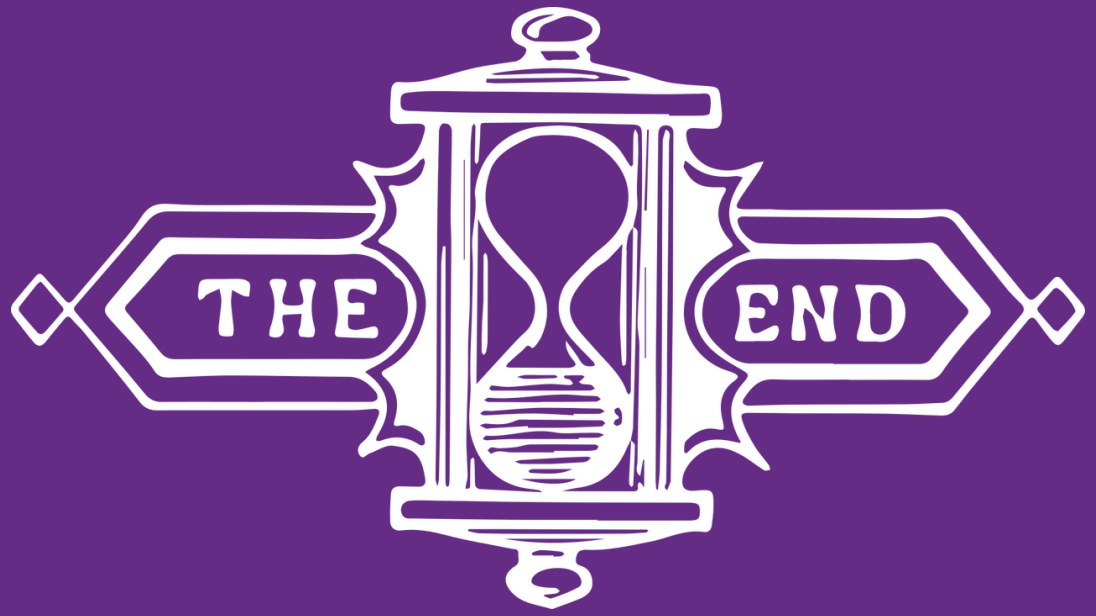
```
from my_package.my_module import *
from my_package.my_second_module import *
```

    → în script importăm obiectele din pachet, fără a fi nevoiti să cunoastem adevărata sursă a lor.

```
from my_package import my_var as tagged_var, my_func, my_second_var

if __name__ == '__main__':
    print('Use another module variable: {}'.format(tagged_var))
    print('Use another module variable: {}'.format(my_second_var))
    print('Call another module function: {}'.format(my_func()))
```

Module și pachete

- Conform PEP 8:
  - numele modulelor trebuie să fie scurte, să conțină doar litere mici, iar cuvintele să fie despărțite de underscore pentru o lizibilitate mai bună (acolo unde este cazul).
  - numele pachetelor trebuie să fie scurte și să conțină doar litere mici. Folosirea underscore-ului nu este indicată.
- Fiind vorba de naming, acestea trebuie să fie intuitive.



Vă mulțumesc!