Python

3. gyakorlat

Strohmayer Ádám

Agenda

- Bitműveletek
- Vezérlési szerkezetek
- Listagenerátorok
- Függvények alapjai

Bitenkénti műveletek

Logikai műveleteink: and, or, not

Bitenkénti műveleteink:

- **~** : negáció
- **&** : bitenkénti és
- **|** : bitenkénti vagy
- **^** : kizáró vagy (XOR) (vagy egyik vagy másik)
- **x << y, x >> y**: bitenként eltolások (x-et y-nal)

Bitenkénti műveletek

a	b	~b	a&b	a|b	a^b
1	1	0	1	1	0
1	0	1	0	1	1
0	1	0	0	1	1
0	0	1	0	0	0

Alacsony precedenciájúak!

```
>>> a = 2
>>> bin(a)
'0b10'
>>> b = 1
>>> bin(b)
'0b1'
>>>
```

Helyi értékenként számoljuk!

Kérdések

Mik lesznek a bitműveletek eredményei?

```
>>> a, b = 5, 6
>>> print(bin(a), bin(b))
0b101 0b110
>>>
```

- a & b
- a | b
- a ^ b
  - b ^ a
- b & 1
- a = a^b
  - b = a^b
  - a = a^b
- a << 1
- b >> 1
- a >> 1
- a | 1
  - b | 1
- a ^ a
  - b ^ 0

Vezérlési szerkezetek

- if, else, elif
- while
- for
  - range

if, else, elif

Elágazások

Formája:

```
if condition1 and condition2:
   #csináld ezt, ha igaz C1, C2
elif condition3:
   #különben ezt, ha igaz C3
else:
   #különben ezt, ha egyik sem igaz
```

Fontos a helyes indentáció!

```
>>> if you.happy and you.know:
...     you.clap_hands()
... else:
...     you.become_happy()
```

```
>>> x = 5
>>> print(x) if x == 3 else print("Nem 3")
Nem 3
>>>
```

while

Elől tesztelő, feltételes, pozitív kiértékelésű ciklus

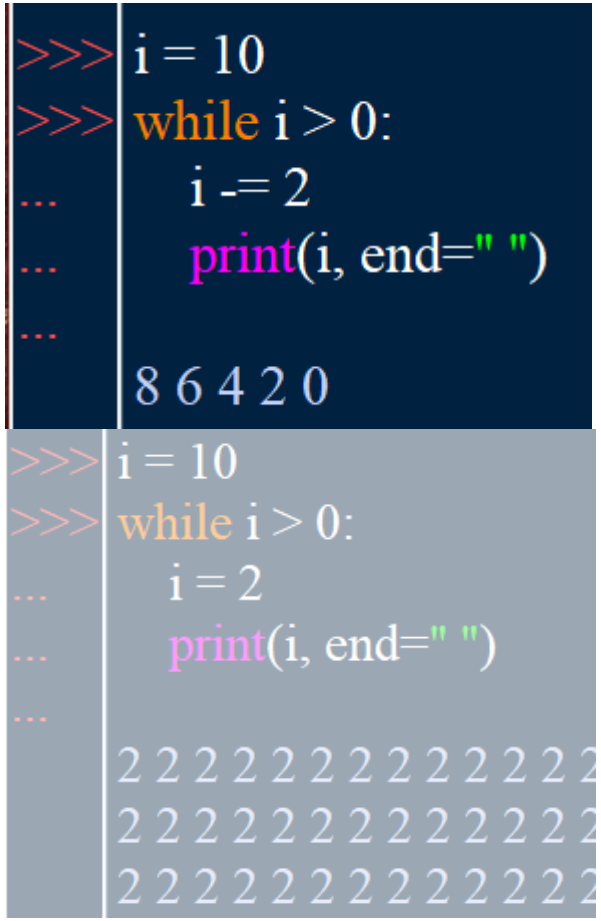
Formája:

```
while condition:
    #csináld ezt
```

Végtelen ciklus:

```
while True:
    #csináld ezt örökké!
```

Megállítása: Ctrl+C



while-**else**?

Else használható a while után!

Formája:

```
while condition:
    #csináld ezt
else:
    #ha (már) nem teljesül a
    #ciklusfeltétel, itt folytatja!
```

while_else_good.py > ...

```
1
2  i = 7
3  while i != 0:
4      i -= 1
5      if i == 10:
6          print("Rossz irány!"); break
7  else:
8      print("Jó irány!")
```

```
>python while_else_good.py
Jó irány!
```

while_else_bad.py > ...

```
1
2  i = 7
3  while i != 0:
4      i += 1
5      if i == 10:
6          print("Rossz irány!"); break
7  else:
8      print("Jó irány!")
```

```
>python while_else_bad.py
Rossz irány!

>
```

for

Hagyományos
és foreach ciklus egyben!

Formája:

for item in **iterable**:
    #csinálj valamit

```
>>> my_dears = ["Mama", "Stan", "Aubrey"]
>>> for dear in my_dears:
...     print(f"Dear {dear}!", end=" ")
...
Dear Mama! Dear Stan! Dear Aubrey!
>>>
```

```
>>> person = {"name" : "Reginald", "job" : "lookout"}
>>> for key, value in person.items():
...     print(f"What is your {key}?  {value}.")
...
What is your name?  Reginald.
What is your job?  lookout.
>>>
```

```
>>> lst_odd = [1, 3, 5]
>>> lst_even = [2, 4, 6]
>>> for x, y in zip(lst_odd, lst_even):
...     print(x, y, end=" ")
...
1 2 3 4 5 6
>>>
```

range

Lusta kiértékelésű függvény!

for ciklusban használatos.

Önmagában csak függvény!

Szintaxis:

range(stop)

range(start, stop)

range(start, stop, lépés)

matematikailag: [start, stop)

```
>>> lst = ["ecc", "pecc"]
>>> for i in range(0, 2):
...     print(lst[i])
...

ecc
pecc
```

```
>>> for i in range(6):
...     print(i, end=" ")
...

0 1 2 3 4 5
>>>
```

```
>>> range(60, 50, -1)
range(60, 50, -1)
>>> list(range(60, 50, -1))
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51]
>>>
```

```
>>> 56 in range(50, 60, 3)
True
>>>
```

List comprehension

Ciklusok besűríthetőek listákba!

```
>>> lst = []
>>> for i in range(6):
...     lst.append(i)
...
>>> lst
[0, 1, 2, 3, 4, 5]
>>>
```

```
>>> lst = [i for i in range(6)]
>>> lst
[0, 1, 2, 3, 4, 5]
>>>
```

```
>>> grades = [("Reginald", 5), ("Adam", 2), ("Wick", 4)]
>>> good_students = [name for name, grade in grades if grade >= 4]
>>> good_students
['Reginald', 'Wick']
>>>
```

Ciklusok sajátosságai

Változók láthatósága más egy ciklusnál,
Más egy listagenerálásnál…

```
>>> x = 5
>>> for x in range(100):
...     pass #semmit nem csinál
...
>>> x
99
```

Fontosabb kulcsszavak:

Break – kilép a ciklusból

Continue – következő iteráció

Pass – nem csinál semmit - **placeholder**

```
>>> x = 5
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
5
>>>
```

Sebességbeli különbség

```
>>> import timeit
>>> def normal():
...     numbers = []
...     for i in range(100000):
...         numbers.append(i * 30)
...
>>> def compre():
...     numbers = [i * 30 for i in range(100000)]
```

Python 3.8

```
>>> timeit.timeit(normal, number = 1000)
17.1063761
>>> timeit.timeit(compre, number = 1000)
11.8290219
>>>
```

Python 3.12

```
>>> timeit.timeit(normal, number = 1000)
10.80944019999879
>>> timeit.timeit(compre, number = 1000)
9.262127800000599
```

Python 3.13

```
>>> timeit.timeit(normal, number = 1000)
9.769449100000202
>>> timeit.timeit(compre, number = 1000)
8.82763039999918
```

List comprehension - kérdés

```
>>> text = """And the earth was without form and void,
...     and darkness was upon the face of the deep."""
>>> x = [[ x for x in line.split() if len(x) < 3] for line in text.split("\n")]
>>>
```

```
[print(x) for x in [f"{" ".join([f"{x*y:3}" for y in range(1,11)])}" for x in range(1, 11)]]
```

csúnya!

Függvények alapjai

Elsőosztályú objektumok!

Névtelen és nevesített függvények is vannak!

```
def function_name(param1 : int, param2 : str) -> int:
    """function description"""
    return param1+param2
```

Névtelen függvény

lambda param1, param_n : expression

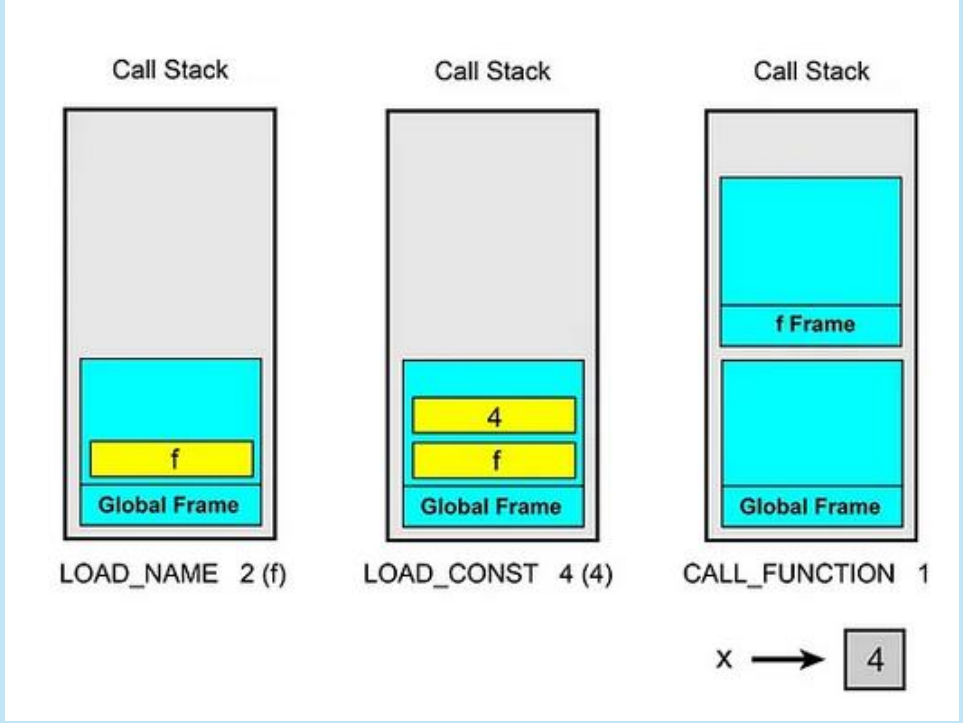
```
>>> greet = lambda name : f"Hi {name}!"
>>> greet("Luke")
'Hi Luke!'
>>>
```

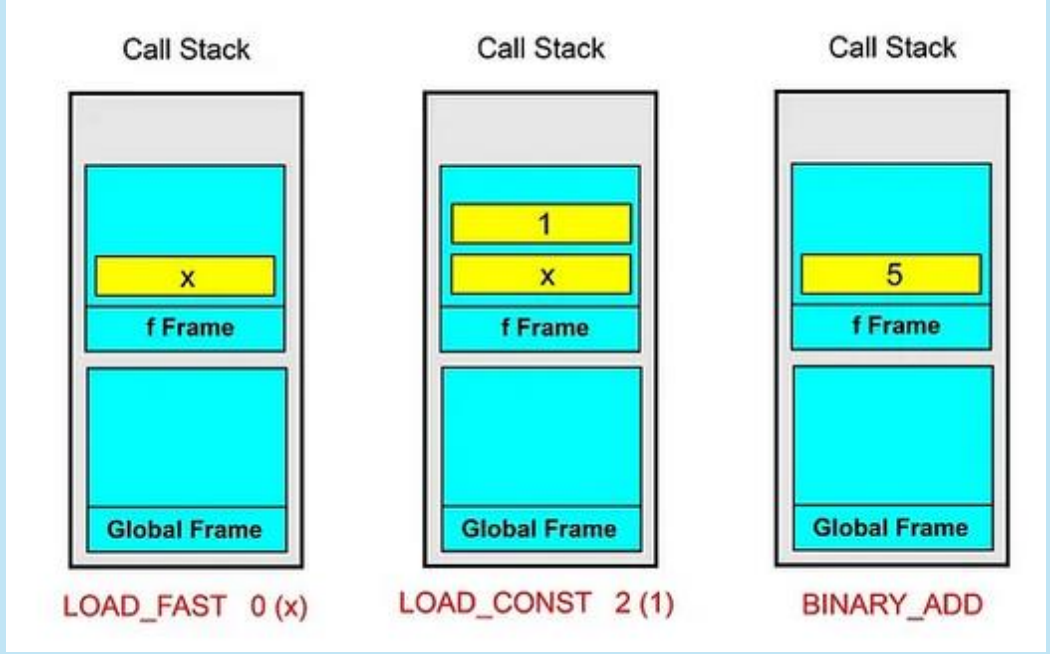
```
>>> hypotenuse = lambda a, b : (a ** 2 + b ** 2) ** 0.5
>>> hypotenuse(3, 4)
5.0
>>>
```

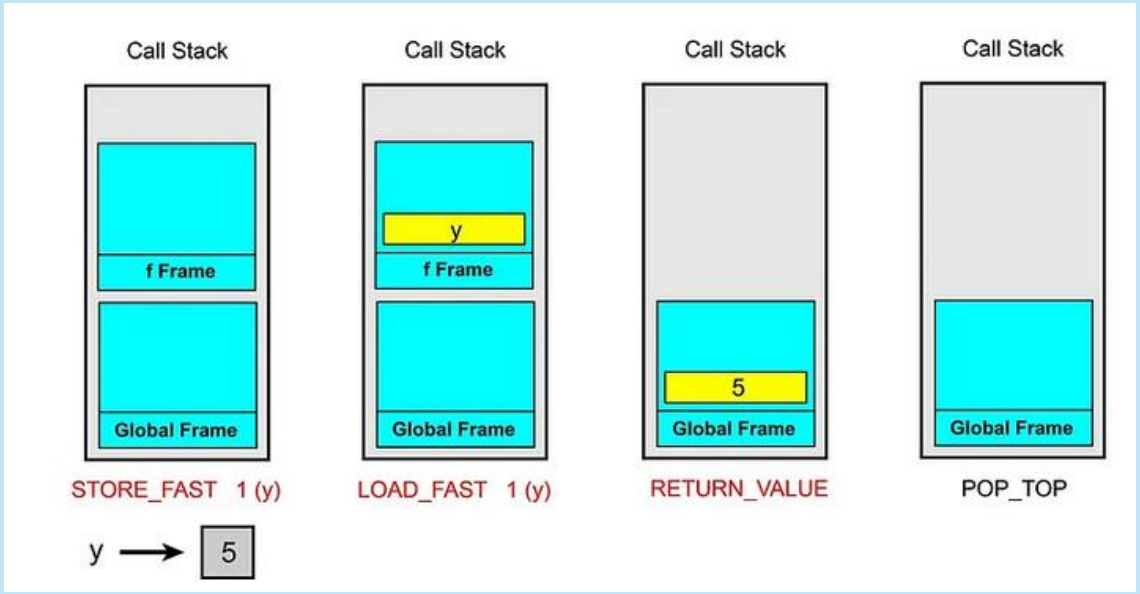
Csak kisebb számításoknál, magasabb rendű függvényekben használjuk!

Függvények a háttérben

- Deklaráláskor eltárolódik a memóriában
- Meghíváskor **saját stack frame**
  - **végrehajtási verembe kerül a keret**
  - Kiértékelés után visszatérés értékkel – a keret eltűnik a veremből
- Végén megkapjuk a kiszámolt értéket!








```
>>> def f(x):
...     y = x + 1
...     return y
...
>>> f(4)
5
```

https://medium.com/towards-data-science/understanding-python-bytecode-e7edaae8734d

Scope - láthatóság

Függvény előtt definiált változók láthatóak.

Ugyanolyan nevű változók a függvényben eltakarják a külső változókat!

Kikerülése: **global**

```
>>> x = 10
>>> def try_global():
...     x = 25
...
>>> try_global()
>>> x
10
>>>
```

```
>>> x = 10
>>> def try_global_2():
...     global x
...     x = 25
...
>>> try_global_2()
>>> x
25
>>>
```

Ne használjuk túl a globalt!

LEGB

Local-Enclosing-Global-Built-ins – ilyen sorrendben keresi a fordító a változóinkat!

```
>>> x = 5
>>> def f1():
...     x = "f1"
...     def f2():
...         x = "f2"
...         print(x)
...     f2()
...
>>> f1()
```

```
>>> x = 12
>>> def f():
...     print(x)
...
>>> f()
12
```

```
>>> x = 13
>>> def f2():
...     print(x)
...     x = 5
...
>>> f2()
```

Scope, closure

Closure eltárolja egy függvényben egy változó állapotát!

```
>>> def power(x):
...     def base(y):
...         return y**x
...     return base
...
>>> power2 = power(2)
>>> print(power2(5))
25
>>> print(power2(10))
100
```

```
>>> def hidden_list():
...     x = [1, 2] #itt lokális, kívülről nem látszik!
...     def printer():
...         nonlocal x
...         x += [1]
...         print(x)
...     return printer
...
>>> closure = hidden_list()
>>> closure()
[1, 2, 1]
>>> closure()
[1, 2, 1, 1]
```

Mi történt eddig?

- Beszéltünk a vezérlési szerkezetekről.
- Megismertük a listageneralást (list comprehension).
- Megismerkedtünk a függvények alap szerkezetével és a lambda függvénnyel.
- Beszéltünk a láthatóságról és a closure jelenségéről.

Feladatok Canvasben!

Köszönöm a figyelmet!