Python

Van a file editor window és az interactive shell

Alap operációk

| **Operator** | **Operation** | **Example** | **Evaluates to . . .** |
| --- | --- | --- | --- |
| \*\* | Exponent | 2 \*\* 3 | 8 |
| % | Modulus/remainder | 22 % 8 | 6 |
| // | Integer division/floored quotient | 22 // 8 | 2 |
| / | Division | 22 / 8 | 2.75 |
| \* | Multiplication | 3 \* 5 | 15 |
| - | Subtraction | 5 - 2 | 3 |
| + | Addition | 2 + 2 | 4 |

Common Data Types

| **Data type** | **Examples** |
| --- | --- |
| Integers | -2, -1, 0, 1, 2, 3, 4, 5 |
| Floating-point numbers | -1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25 |
| Strings | 'a', 'aa', 'aaa', 'Hello!', '11 cats' |

’a’+’b’= ’ab’

’a’+1= error

’a’\*5= ’aaaaa’

’a’\*’b’= error

’a’\*5.0= error

Változó (variables)

>>>a = 2

>>>a

2

>>>b = 3

>>>a + b

5

>>>a = a+2

>>>a

4

>>>a = ’Szia’

>>>a

’Szia’

>>>a = ’bye’

>>>a

’bye’

Valid and Invalid Variable Names

| **Valid variable names** | **Invalid variable names** |
| --- | --- |
| current\_balance | current-balance (hyphens are not allowed) |
| currentBalance | current balance (spaces are not allowed) |
| account4 | 4account (can’t begin with a number) |
| \_42 | 42 (can’t begin with a number) |
| TOTAL\_SUM | TOTAL\_$UM (special characters like $ are not allowed) |
| hello | 'hello' (special characters like ' are not allowed) |

Első program

➊ # This program says hello and asks for my name.  
  
➋ print('Hello, world!')  
   print('What is your name?')    # ask for their name  
➌ myName = input()  
➍ print('It is good to meet you, ' + myName)  
➎ print('The length of your name is:')  
   print(len(myName))  
➏ print('What is your age?')    # ask for their age  
   myAge = input()  
   print('You will be ' + str(int(myAge) + 1) + ' in a year.')

1, A #-es szöveg nem érdekli a programot (commentelésre használják), a kihagyott sorok sem zavarják

2, A **print()** függvény azt jelenti hogy nyomtassa ki a képernyőre ami a () közt van

3, Az **input()** függvény arra vár hagy írjunk valamit a billentyűzettel és nyomjunk egy entert

5,A **len()** függvénynek adhatunk egy string-et vagy egy változót amiben egy string van ezekben lévő karakterek számát kiszámolja

>>>len(’a’)

1

>>>len(’Hello’)

5

mint ezek a példák len(myAge) kap egy interger-t

de:

>>>print(’Én’ + 18 + ’éves vagyok’) = error mert intergert és stringet nem lehet össze adni, így string verzióban (pl.: myAge itt) kell használni az intergert

A str(), int() és float() függvényk

Ha egy intergert mint a 10-et akarunk print() egy str-rel akkor az str()-nek adhatunk egy interger értéket és nem kapunk errort

>>>str(10)

’10’

>>>print(’Én ’ + str(10) + ’ éves vagyok.’)

**str(),** **int()** és **float()** kifejezni a saját formáját a string-nek, interger-nek, és floating-point-nak

>>> str(0)  
'0'  
>>> str(-3.14)  
'-3.14'  
>>> int('42')  
42  
>>> int('-99')  
-99  
>>> int(1.25)  
1  
>>> int(1.99)  
1  
>>> float('3.14')  
3.14  
>>> float(10)  
10.0

A str() hasznos amikor egy szövegben egyszerre egy floating-point-ot vagy egy interger-t akarunk egy stringben

Az int() hasznos ha van egy változód (ami string) amivel számolásokat akarsz csinálni:

>>>a = input()  
101  
>>>a  
'101'

Itt az a nem az interger 101-el egyenlő hanem a string ’101’-el

Ha matekot akarsz csinálni az a-ban lévő értékkel akkor az int()-et kell használnod és így az interger verzióját kapod meg

>>> a = int(a)  
>>> a  
101

Mostmár lehet az a válzozót intergerként használni

>>> a \* 10 / 5  
202.0

De ha olyan értéket adsz az int()-nek ami nem interger akkor errort kapsz

>>> int(’1.1’)

error

>>>int(’tíz’)

error

Az int() függvény arra is jó ha lekell floating-point számokat kerekíteni

>>> int(7.7)  
7  
>>> int(7.7) + 1  
8

**TEXT AND NUMBER EQUIVALENCE**

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

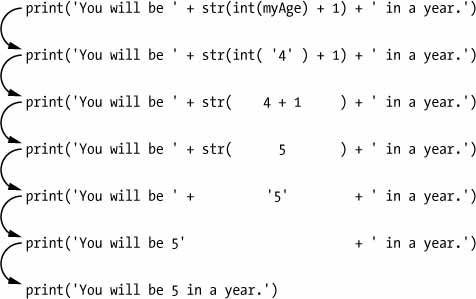
>>> **42 == '42'**  
False  
>>> **42 == 42.0**  
True  
>>> **42.0 == 0042.000**  
True

Python makes this distinction because strings are text, while integers and floats are both numbers.

➏ print('What is your age?') # ask for their age  
   myAge = input()  
   print('You will be ' + str(int(myAge) + 1) + ' in a year.')

Az előző kódban három sorban is használtuk a str() és int() függvényt

A myAge változó megkapta az input() értékét. Az input() az mindig stringet ad emiatt kell az int(myAge) –et használnunk hogy hozzá lehessen adni az 1-et, így kapjuk az int(myAge) + 1) függvényt. Ennek az eredményét visszaváltoztjuk string –é a str() függvényval és így kapjuk a str(int(myAge) + 1) függvényt. Ezt hozzáadjuk a ’You will be ’ és az ’ in a year.’ –hoz



Round()

A **round()** függvény a legközelebbi egész számra kerekít, ha a kerekítendő szám ugyananyira van két egész számtól akkor mindig a párosat választja.

>>> round(1.25)

1

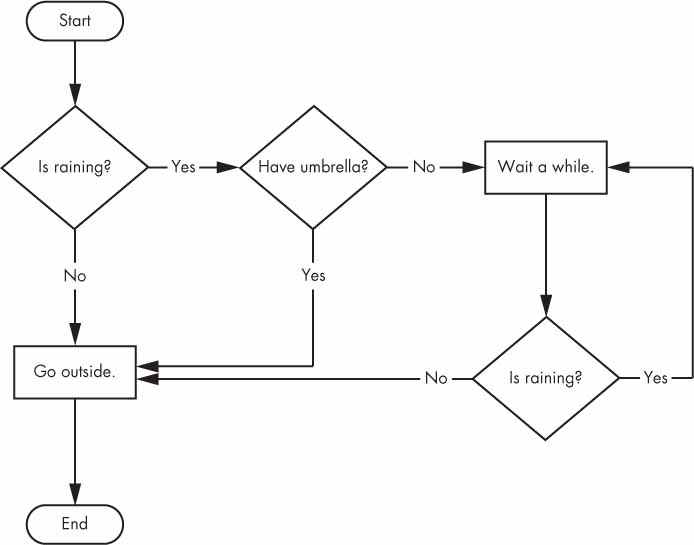
>>> round(0.5)

0

>>> round(-0.5)

0

Flow control statements



Boolean values

Csak két értéke lehet (True, False)

Nagybetűvel kezdődik

Nincs ’’

➊ >>> a = True  
   >>> a  
   True  
➋ >>> true  
   Traceback (most recent call last):  
     File "<pyshell#2>", line 1, in <module>  
       true  
   NameError: name 'true' is not defined  
➌ >>> True = 2 + 2  
   SyntaxError: can't assign to keyword

Mint bármilyen más érték lehet változóban tárolni (1). Ha nem helyesen írod (2) vagy változónak próbálod használni errort kapsz

Comparision operators

Kettő értéket összehasonlítunk és kapunk belőle egy boolean értéket.

| **Operator** | **Meaning** |
| --- | --- |
| == | Equal to |
| != | Not equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |

Ezeknek a végeredménye True vagy False attól függően, hogy milyen értékeket adunk nekik

>>> 42 == 42  
True  
>>> 42 == 99  
False  
>>> 2 != 3  
True  
>>> 2 != 2  
False

Az == és a != minden fajta adat típussal műkődik

   >>> 'hello' == 'hello'  
   True  
   >>> 'hello' == 'Hello'  
   False  
   >>> 'dog' != 'cat'  
   True  
   >>> True == True  
   True  
   >>> True != False  
   True  
   >>> 42 == 42.0  
   True  
 >>> 42 == '42'  
   False

Az interger és a floating-point mindig nem enyenlő lesz a string-el.

A <, >, <=, és >= viszont a string-nél nem működnek rendesen

   >>> 42 < 100  
    True  
    >>> 42 > 100  
    False  
    >>> 42 < 42  
    False  
    >>> eggCount = 42  
 ➊ >>> eggCount <= 42  
    True  
    >>> myAge = 29  
 ➋ >>> myAge >= 10  
    True

Az == azt kérdi hogy egyenlőek-e az állítások míg, az = az eredményre kérdez rá

Boolean operators

And, or, not

Binary Boolean operators:

Az and és az or mindig 2 Boolean értéket vesznek ezért hívjuk őket bináris operátoroknak.

The and Operator’s Truth Table

| **Expression** | **Evaluates to . . .** |
| --- | --- |
| True and True | True |
| True and False | False |
| False and True | False |
| False and False | False |

The or Operator’s Truth Table

| **Expression** | **Evaluates to . . .** |
| --- | --- |
| True or True | True |
| True or False | True |
| False or True | True |
| False or False | False |

A nem operátor

A nem csak egy Boolean értéken dolgozik emiatt urinary operátornak hívjuk.

A nem csak szimplán megfordítja az értéket

The not Operator’s Truth Table

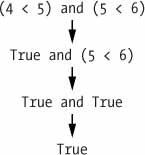
| **Expression** | **Evaluates to . . .** |
| --- | --- |
| not True | False |
| not False | True |

>>> not False

True

>>> not not True

True

Mixing Boolean and Comparison operators

>>> (4 < 5) and (5 > 6)

True

>>> (1 > 2) and (4 > 3)

False

>>> (1 == 2) or (2 == 2)

True

>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 \* 2 == 2 + 2

True

A Boolean operátoroknak is van műveleti sora:

1.not

2.and

3.or

A flow control részei

Flow control utasításek (statements) általában egy feltétel (condition) nevű részel kezdődnek, ezt követi a clause

Conditions

A Boolean kifejezések (expressions) amiket eddig írtunk mindet feltételnek nevezhetjük

Expression = Condition (Ezt a nevet használjuk flow controlban)

A Feltételeknek a végeredménye mindig True vagy False

Blocks of code

Blockokba lehet sorolni a python kódot

3 szabály van:  
1. A block akkor kezdődik amkor a behúzás nő

2. A blockokban lehetnek blockok

3. A block akkor végződik, amikor a behúzás visszament a kezdet szintéig

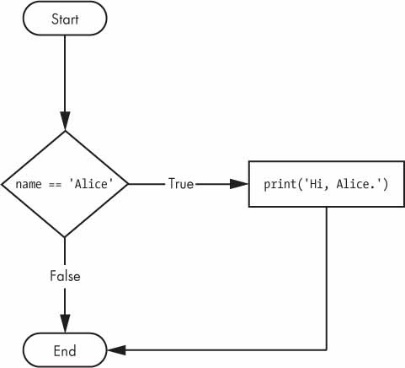
 name = 'Mary'  
   password = 'swordfish'  
   if name == 'Mary':  
     ➊ print('Hello, Mary')  
        if password == 'swordfish':  
         ➋ print('Access granted.')  
        else:  
         ➌ print('Wrong password.')

1, Az első block

2,Egy második block az előtte lévőben és csak egy sor hosszú

Az **if** utasítás

A leggyakoribb statement és ha az **if** statement True akkor az alatta lévő clause megtörténik ha visszont False akkor átugorja

4 részből áll:

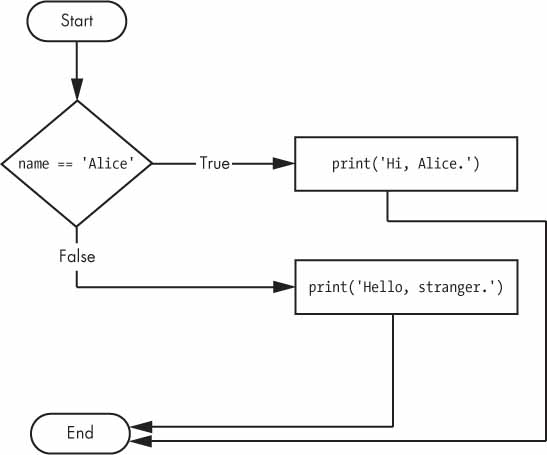
* az if kulcsszó
* egy condition (kifejezés ami True or false)
* egy kettőspont (:)
* az if utáni sorban egy új block a clause-al

if name == 'Alice':  
     print('Hi, Alice.')

Az **else** utasítás (statement)

A clause-t követheti, csak ha az if statement False

3 részből áll:

* Az else kulcsszó
* Egy kettőspont
* az else utáni sorban egy új block a clause-al (else clause a neve)

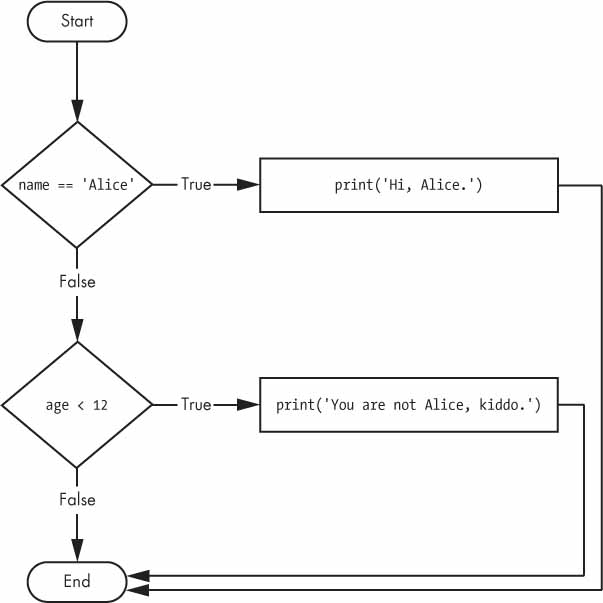
if name == 'Alice':  
     print('Hi, Alice.')  
 else:  
     print('Hello, stranger.')

Az **elif** statement

Még az else és az if-nél csak egy clause történhet, az elif-nél rengeteg clause is lehet.

elif= else if

Mindig if-t vegy elif-t követ

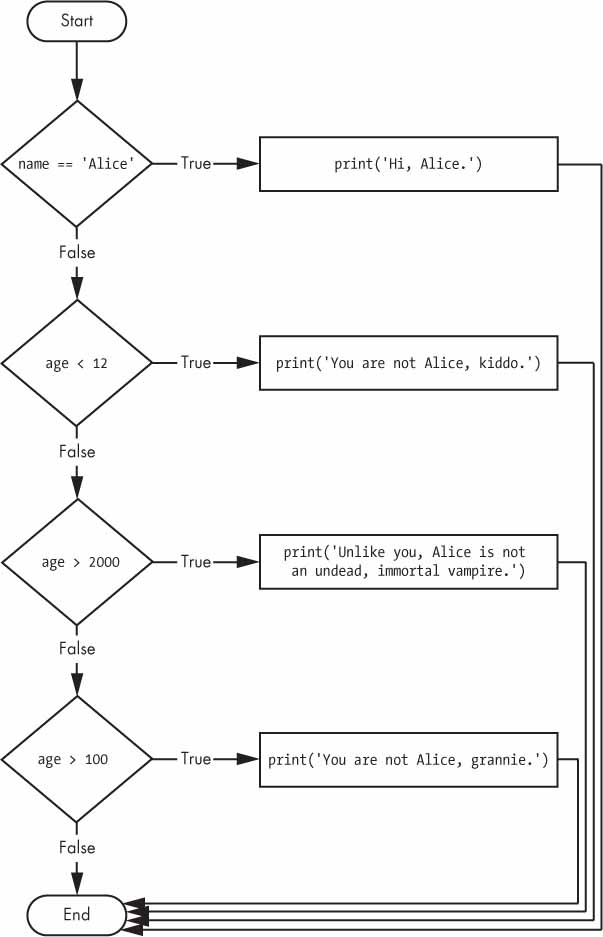
* az elif kulcsszó
* egy condition (kifejezés ami True or false)
* egy kettőspont (:)
* az elif utáni sorban egy új block a clause-al (elif clause a neve)

if name == 'Alice':  
     print('Hi, Alice.')  
 elif age < 12:  
     print('You are not Alice, kiddo.')

Akkor tüzel az elif clause, ha age < 12 True és name == ’Alice’ False. Ha mind a kettő false akkor semelyik clause sem tüzel.

Az elif-nél nem garantált hogy hogy bármelyik clause meg fog történni.

Egy elif láncnál csak 1 vagy 0 clause fog tüzelni

Amint a láncban az egyik elif True akkor átugorja az összes utána lévőt

name = 'Máté'  
age = 3000  
  
if name == 'Márk':  
 print('Szia, Márk!')  
  
elif age < 12:  
 print('Nem Márk vagy, kiddo.')  
  
elif age > 2000:  
 print('Márk nem egy élőhalott, halhatatlan vámpír.')  
  
elif age > 100:  
 print('Nem Márk vagy, nagyi.')

Ezt a programot else-el is átírhatjuk:

name = 'Máté'  
age = 3000  
  
if name == 'Márk':  
 print('Szia, Márk!')  
  
elif age < 12:  
 print('Nem Márk vagy, kiddo.')  
  
else:  
 print('Se Márk se egy kis kiddo vagy.')

Szabályok:

* Mindig csak egy if van
* Ha kell elif az az if-t kövesse
* Zárjunk else-el

while Loop utasítás

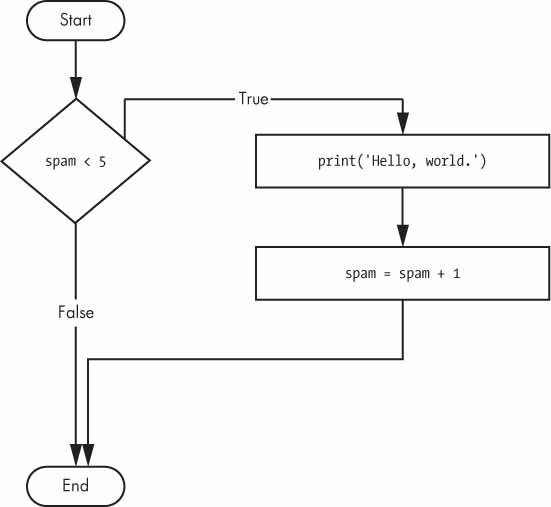
egy blockot egymás után többször lehet futtani a while statement-el

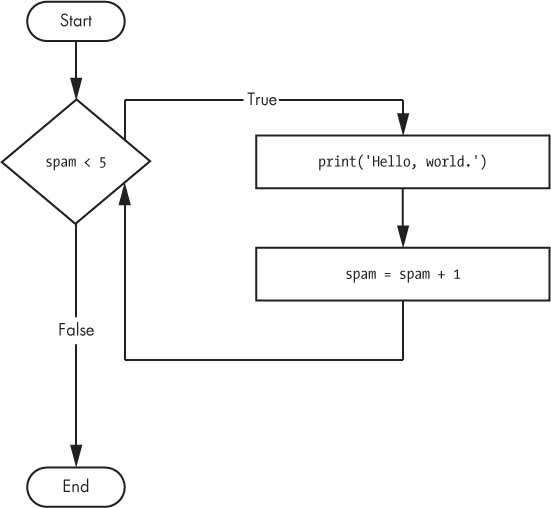
Addik fog futni amig a statement True

4 részből áll:

* az if kulcsszó
* egy condition (kifejezés ami True or false)
* egy kettőspont (:)
* a while utáni sorban egy új block a clause-al (while clause)

Egy kód if-el:

 spam = 0  
 if spam < 5:  
     print('Hello, world.')  
     spam = spam + 1

Egy kód while-al:

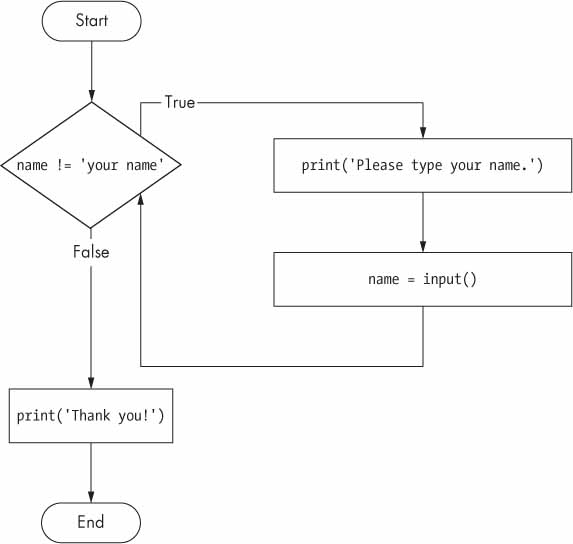
spam = 0  
 while spam < 5:  
     print('Hello, world.')  
     spam = spam + 1

Ha az if-eset futtatod csak egyszer írja a szöveget míg a while-osnnál ötször

Az if-es kód csak lecsekkolja a feltételt (condition)egyszer hogy True-e. A while loop azomban addig csinálja amíg False nem lesz

Egy idegesítő while loop

name = ''  
while name != 'your name':  
 print('Please type your name.')  
 name = input()  
  
print('Thank you!')



Először a program a name variable-t egy üres string-re állítja, amiatt hogy a name != ’your name’ True legyen és a program megkezdje a while clause-át

A kód megkér hogy írd be a nevedet amit belerak a name változóba, mivel ez az utolsó sor a program visszalép a while loop elelyére, ha a name nem egyenlő a ’ your name’ string-el akkor a condition True és újrakezdődik a while clause

De amint beírod hogy ’your name’ a utasítás False lesz és ahelyett hogy újrakezdené átskippeli a program és a többi részét kezdi futtatni

Ha sose írod be hogy ’your name’ a condition öröké False lesz és ismétlődik

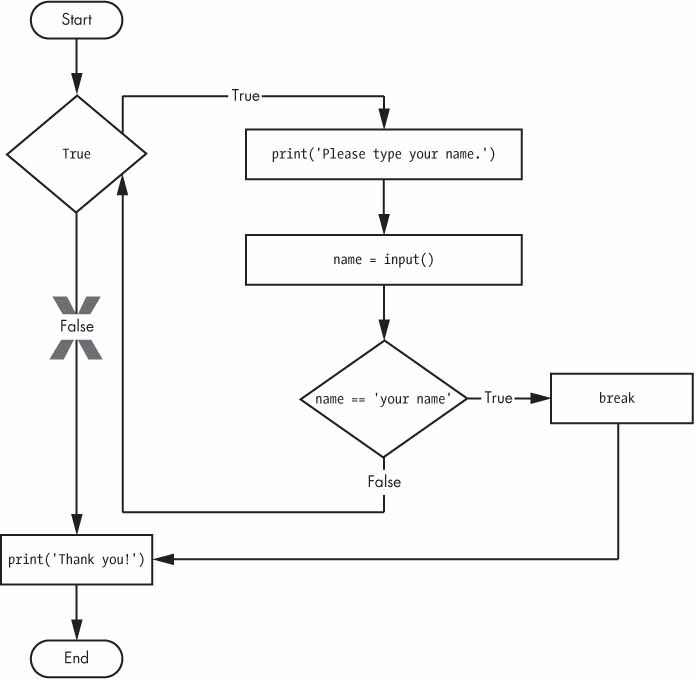
Break utasítás

Van egy égérút amivel kilehet törni egy while loop clause-ából korán

Ha a kód eléri a **break** statementet, akkor röktön kilép a while loop clause-ábol

Egy idegesítő while loop break-el

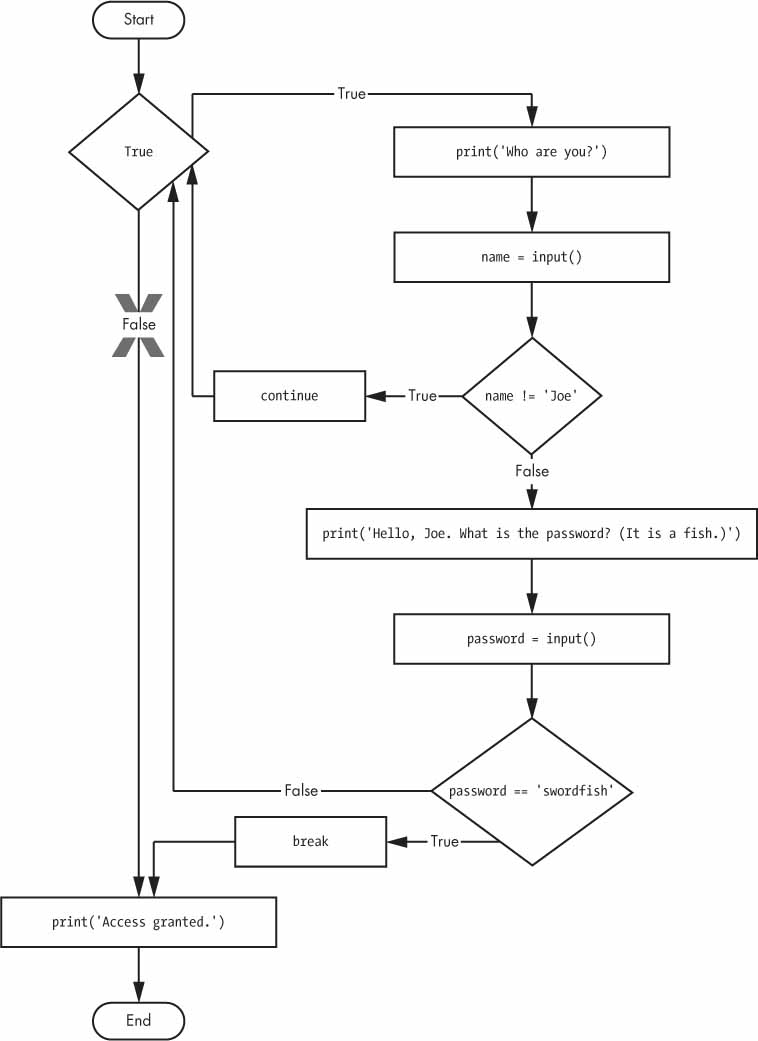
while True :  
 print('Please type your name.')  
 name = input()  
  
 if name == 'your name' :  
 break  
  
print('Thank you!')



Az első sor készít egy végtelen loop-ot (egy while lope aminek a codition-ja mindig True). Amikor a program megkezdi ezt a loop-ot csak akkor fogja elhagyni, ha a break statement tüzel.

Mint múltkor a program megkér, hogy írd be a your name-t. Most viszont egy if utasítás nézi meg hogy name == ’your name’. Ha ez igaz, akkor a break statement fut és a program kilép a loop-ból és print()-el. Ha viszont átlépi a program, mert hamis akkor tovább folytatódik a kör, amíg a helyes kódot be nem írlyuk, hiszen a condition-ja (feltételje) a while loop-nak a True Boolean érték.

Continue statement

Mint a break kifejezéseket, a **continue** statement-eket loop-okban használjuk

Amikor a program egy continue-hoz ér röktön visszaugrik a loop kezdetéhez és újrakérdi a loop condition-ját

while True:  
 print('Who are you?')  
 name = input()  
  
 if name != 'Joe':  
 continue  
  
 print('Hello, Joe. What is the password? (It is a fish.)')  
 pasword = input()  
  
 if pasword == 'swordfish':  
 break  
  
print('Acces granted.')

Ha a felhasználó bármilyen más nevet ír be, mint Joe a continue miatt a kód visszaugrik a loop kezdetére. Amint a felhasználó átjut az if kifejezésen, a kód kéri a jelszót. Ha a jelszó, amit beírunk swordfish, akkor a break statement fut, és a kód kiugrik a while loop-ból hogy print-eljen, ha nem ez történik, akkor visszaugrik a loop elejére

Az igazkás és hamiskás értékek

Amikor condition-okban (feltétlekben) használjuk a 0, 0.0, ’ ’ ezeket a program False-nak gondolja, míg mindem más értéket True-nak gondol

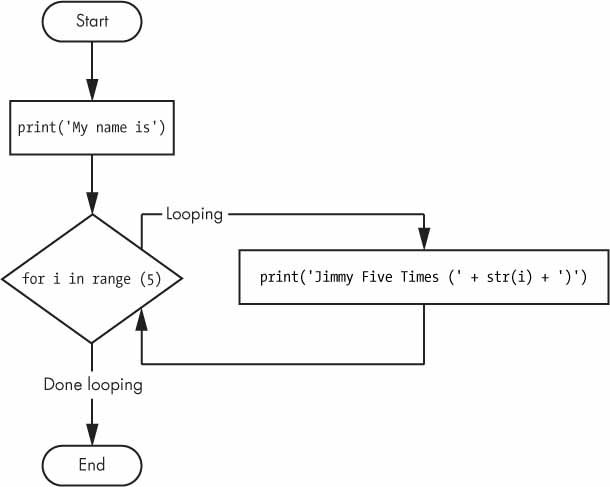
name = ''  
  
while not name :  
 print('Enter your name:')  
 name = input()  
  
print('How many guests will you have?')  
n\_of\_g = int(input())  
  
if n\_of\_g:  
 print('Be sure to have enough room for all you guests.')  
  
print('Done')

Ha egy üres string-et rakunk a name-nek akkor a while statement condition-ja True lesz, és a program újra kérdez. Ha az értéke az n\_of\_g-nek nem 0 akkor a feltétel True és a program print-el.

for loop-ok és a range() function

A while loop addig fut amíg a utasításe True, de a **for** loop és a **range()** függvény csak egy megadott számszor fut.

6 részből áll:

* A for kulcsszó
* egy változó
* Az in kulcsszó
* A range() függvény maxiumum 3 interger-rel
* Egy kettőspont
* a for utáni sorban egy új block a clause-al (for clause)
* print('My name is')  
  for i in range(5) :  
   print('Jimmy five times (' + str(i) + ')')

A kód a for loop clause-ában 5-ször fut. Mi után fut egyszer a clause a program visszamegy az elejére és az i-hez hozzá ad egyet.

breake-t és continue-t tudunk for loop-okban is használni. A continue továbblép a for loop számlálójában, mintha befejezte vokna azt a kört.

t = 0  
p = ''  
  
while not p:  
 print('Meddig akarunk számokat összeadni?')  
 p = input()  
  
p = int(p) + 1  
  
for n in range(p):  
 t = t + n  
  
print(t)

A prigram úgy kezd, hogy a t 0—ra van állítva. A while loop megkérdi, hogy mennyit ér a p, ehez hozzá kell adni egyet hogy pont anyiszor futson a for program, mint amit beírtunk. A for loop rárakja az n-re a jellenlegi p értékét majd ezt hozzáadja a p-hez, amint a számig összead, amit beírtunk a program print-eli a t értékét.

A Starting (indulás), Stooping (megéllás) és Stepping (ugrás) Arguments (argumentumek) a range()-hez

Vannak olyan function-ok amiket meglehet változtatni argumentumekkel, vesszővel elválasztva, a range() is egy ilyen. Emiatt megváltoztathatod az interger-t amit adsz neki, így egy másik számmal is kezdhetsz mint nulla.

for i in range(12, 16):  
 print(i)

Az első lesz, amivel kezd és az utolsó lesz, ahol megáll (de nincs benne)

12

13

14

15

Három argumentet is használhatunk a range()-nél. Az első kettő lesz a start és a stop, a harmadik meg a step lesz. A step az az amennyivel nő a szám minden iteráció után.

for i in range(0, 10, 2):  
 print(i)

Szóval a range (0, 10, 2) nullától nyolcig fog számolni kettesével

0

2

4

6

8

Még mínusz értéket is adhatunk bármelyik értéknek, ilyenkor a step visszafele lépeget.

for i in range(5, -1, -1):  
 print(i)

Ez a következőt fogja print()-elni:

5

4

3

2

1

0

Importing modules (modulok importálása)

Minden Python programnak van alap beépített függvényi, mint a print(), input(), len(), stb… De a Python jön még egy modul készlettel, amit úgy hívnak, hogy standard library. Mindben function-ok vannak, amiket használhatsz a kódodban (a math modulban matematikához függő dolgok vannak, a random-ban meg random számokhoz függő dolgok vannak.

2 részből áll:

* Az import kulcsszó
* A modul neve (több modul is lehet, vesszővel elválasztva)

Amikor importálsz egy modul használhatod a menő függvényit. A random modulnál például használhatjuk a random.randint() függvényt.

import random  
  
for i in range(5):  
 print(random.randint(1, 10))

Amikor futtattjuk valahogy így fog kinézni

6

8

10

8

9

A random.randint() function ad egy random interger-t a kettő interger közt, amiket adtál neki. Mivel a randint() a random mudulban van, emiatt először elé kell írnod, hogy random., hogy ebben a modulban keresse.

from import statements (állítás)

Egy alternatív módja az import-nak.

4 részből áll:

* a from kulcsszó
* a modul neve
* az import kulcsszó
* egy csillag

pl.: from random import \*

Így nem kell mindig beírni, hogy random., de így nehezebb a kódot átolvasni, szóval inkább használjuk az import random formáját az állításnak.

Egy program korai befejezése sys.exit()-el

Egy program mindig véget ér az utolsó utasítás után, de ha azt akarod, hogy ez előbb legyen akkor a sys.exit() fonkciót használhatod. Mivel ez a sys modulban van először azt import-álni kell

import sys  
  
while True:  
 print('Type exit to exit.')  
 resp = input()  
  
 if resp == 'exit' :  
 sys.exit()  
  
 print('You typed ' + resp + '.')

Ebben a programban van egy végtelen loop break nélkül. Az egyetlen módja hogy vége legyen az az, hogyha a végrehajtás eléri a sys.exit()-et. Amikor resp egynlő exit-el akkor fut ez a sor.

Egy rövid program: Találd ki a számot

import random  
  
titkn = random.randint(1, 20)  
  
print('Gondoltam egy számra 1 és 20 között.')  
  
for próbák in range (1, 7):  
 print('Tippelj!')  
 tipp = int(input())  
  
 if tipp < titkn:  
 print('Túl kicsit tippeltél.')  
  
 elif tipp > titkn:  
 print('Túl nagyot tippeltél.')  
  
 else:  
 break  
  
if tipp == titkn:  
 print('Ügyes vagy! Eltaláltad a számot ' + str(próbák) + ' próba alatt!')  
  
else:  
 print('Nem találtad el. A szám ' + str(titkn) + ' volt.')

először a program import-álja a random modult goy használhatsa random.randint() function-t, hogy generáljon egy számot 20 és 1 között, ezt a titkn-ben tárolja.

A program elmondja, hogy gondolt egy számra és ad a játékosnak hat próbát, hogy kitalálja. A játékos beír egy tippet és a kód csekkolja egy for loop-ban ami hatszor loop-ol. Először a beírt tippet a program átváltoztatja, egy int()-é, mert az input()-ból mindig string-et kapunk. Ezt az értéket elraktározza egy változóban tipp néven. Utána a kód megnézi, hogy a tipp alacsonyabb vagy magasabb-e mint a gondolt szám, ha igen akkor kap a játékos egy tippet. Ha a szám nem kisebb vagy nagyobb, mint a titkos szám akkor csak egynlő lehet és akkor ki break-el a program a loopbol. Miután vége a for loop-nak a program megnézi, hogy az if-es statement True vagy False és kinyomtatja az ehez való sort.

Egy rövid program: Kő, Papír, Olló

import random, sys  
  
print('KŐ, PAPÍR, OLLÓ')  
  
nyerések = 0  
vesztések = 0  
döntetlenek = 0  
  
while True:  
 print('%s Nyerések, %s Vesztések %s Döntetlenek' %(nyerések, vesztések, döntetlenek) )  
  
 while True:  
 print('Írd be a lépésed: (k)ő (p)apír (o)lló vagy (ki)lépés ')  
 lépés = input()  
 if lépés == 'ki':  
 sys.exit()  
  
 if lépés == 'k' or lépés == 'p' or lépés == 'o':  
 break  
  
 print('Írj k-, p-, o-, vagy ki-t.')  
  
  
 if lépés == 'k':  
 print('KŐ vs...')  
  
 elif lépés == 'p':  
 print('PAPÍR vs...')  
  
 elif lépés == 'o':  
 print('OLLÓ vs...')  
  
  
 rnumb = random.randint(1, 3)  
  
 if rnumb == 1:  
 comp\_l = 'k'  
 print('KŐ')  
  
 elif rnumb == 2:  
 comp\_l = 'p'  
 print('PAPÍR')  
  
 elif rnumb == 3:  
 comp\_l = 'o'  
 print('OLLÓ')  
  
  
 if lépés == comp\_l:  
 print('Döntetlen!')  
 döntetlenek = döntetlenek + 1  
  
 elif lépés == 'k' and comp\_l == 'o':  
 print('Nyertél!')  
 nyerések = nyerések + 1  
  
 elif lépés == 'p' and comp\_l == 'k':  
 print('Nyertél!')  
 nyerések = nyerések + 1  
  
 elif lépés == 'o' and comp\_l == 'p':  
 print('Nyertél!')  
 nyerések = nyerések + 1  
  
 elif lépés == 'k' and comp\_l == 'p':  
 print('Vesztettél!')  
 vesztések = vesztések + 1  
  
 elif lépés == 'p' and comp\_l == 'o':  
 print('Vesztettél!')  
 vesztések = vesztések + 1  
  
 elif lépés == 'o' and comp\_l == 'k':  
 print('Vesztettél!')  
 vesztések = vesztések + 1

Először import-áljuk a random és sys modult, hogy tudjuk használni a random.randint()-et és a sys.exit-et. Emellett felállítunk 3 változót, hogy tudjuk számolni a vesztéseket, nyeréseket és döntetleneket.

Ez a program egy while loop-ot használ egy másik while loop-ban. Az első loop a játék alapja és minden egyes kő papír olló ebben játszódik, míg a második kéri meg a játékost, hogy írja be a lépését és addig loop-ol, amíg be nem írt a játékos egy k-, p-, ó- vagy ki-t. Amikor a játékos kit ír akkor a sys.exit() fut és a program terminálódik. Ha a játékos beírja a k-, p- vagy ó-t akkor ez a belső loop break-el.

A számítógép lépése random szelektálodik. Az 1 és 3 közötti int a rnumb-ban tárolódik. A rnumb alapján megkapja a comp\_1 az értékét.

Aztán a kód összehasonlítja a string-eket a lépés-t és a comp\_1 és kiírja az eredményt a képernyőn. Emellet növeli a nyerések, vesztések, döntetlenek értékét a végeredmény szerint. A legvégén a kód visszaugrik a játék elejére.

Practice questions

1. True, False
2. and, or, not
3. True and True = True, True and False = False, False and True = False, False and False = False; True or True = True, True or False = True, False or True = True, False or False = False; not True = False, not False = True
4. False, False, True, False, False, True
5. ==, !=, <=, >=, <, >
6. Az elsővel értéket adunk valaminek, a ==-vel meg megnézzük, hogy ugyanaz-e a két állítás.
7. Egy kifejezés, aminek az értéke egy Boolean érték.
8. A 2 if és az egy else
9. if spam == 1:  
    print('Hello')  
     
   elif spam == 2:  
    print('Howdy')  
     
   elif spam == 3:  
    print('Greetings!')
10. CTRL-C
11. a break kitör a loop-ból, míg a continue visszamegy az elejére.
12. a range(10) felszámol tízig (9-nél megáll és 0-nál kezd), a range(0, 10) az nullától felszámol tízig, a range(0, 10, 1) egyesével felszámol nullátol tízig.
13. for n in range(1, 11):  
     print(n) n = 1  
      
    while n != 11 :  
     print(n)  
     n = n + 1
14. spam.bacon()

abs() function

A szám abszolút értékét adja vissza

Functions

a function-ok olyanok, mint mini programok egy programban

def hello():  
 print('Howdy!')  
 print('Howdy!!!')  
 print('Hello there.')  
  
hello()  
hello()  
hello()

Az első sor egy def statement, ami definiálja a hello() nevű függvényt. A def utáni block az a függvény teste, ez fut, amikor a függvényt tüzeljük, nem akkor, amikor meghatározzuk.

A hello() sorok a function után a function calls-ok (függvényhívások). A függvényhívás csak a függvény neve egy zárójellel követve, ebben lehetnek argument-ek. Amikor a kód eléri a hívásokat, akkor felugrik a function-hoz és futtatja a benne lévő kódot, amikor ennek vége, akkor visszaugrik a hívás alá.

Mivel ez a program a hello()-t háromszor hívja, emiatt a kód a hello() függvényban háromszor fut.

A legfőbb szándéka a függvényknak az az, hogy csoportosítsunk kódot, amit többször akarunk futtatni, itt a függvény nélkül ugyanazokat a sorokat kellet volna többször kiírni.

def Statements with Parameters

Amikor a print() vagy len() function-t használod, adhatsz nekik értékeket, amiket argumentumeknek (arguments) hívnak, ha a zárójel közé írsz. A saját függvénydat is tudod úgy definiálni, hogy elfogadjon argumentumeket.

def hello(name):  
 print('Hello, ' + name )  
  
hello('Mark')  
hello('Rita')

A hello() függvénynek van egy paramétere, amit úgy hívnak, hogy name. A paraméterek olyan változók, amiknek vannak argumentumei. Amikor egy függvényt hívunk argumentumekkel, az argumentumek a paraméterekben vannak eltárolva. Amikor először hívjuk a hello() függvényt akkor a ’Mark’ argumentumet kapja. Amikor a program belép a function-ba a name paraméter automatikusan átváltozik ’Mark’-á amit aztán kinyomtat a print() statement-ben (utasításben).

A paraméterben az érték elfelejtődik, amikor a függvény visszalép a lehívás mögé. Például, ha a hello(’Mark’) mögé beírtuk volna, hogy print(name) error-t kaptunk volna, mert nincs olyan változó, hogy name. Ez a változó megsemmisül amiután a függvényhívás hello(’Mark’) visszalép.

Define, call, pass, argumentumek, paraméterek

def say\_hello(name):  
 print('Hello, ' + name)  
   
say\_hello('Mark')

Definiálni egy function-t az azt jelenti magyarul, hogy létrehozni, mint egy assignment statement (hazarendelési utasítás), mint a toll = 42, ami a létrehozza a toll változót. A def utasítás meghatározza, definiálja a say\_hello() függvényt. A say\_hello(’Mark’) sor hívja a függvényt és küldi a végrehajtást a függvény elejére. Ez a függvényhívás átadja (passing) a ’Mark’ string értéket a függvénynek.

Az érték, amit a függvénynek küldünk egy függvényhívásban argument-nek (argumentumnek) hívjuk. A ’Mark’ arguement-rt hozzárendeli a program a name helyi változóhoz (local variable). Változok, amiknek vannak argumentumei paraméterek.

Return értékek és return statements (utasításek)

Amikor a len() függvényt hívod és adsz neki egy argumentumet, mint a ’Hello’, azt az intergert fogod kapni, ami a szó hossza, itt az 5. Általánosan az érték amit visszakapsz egy függvényból az a return érték.

Amikor egy függvényt készítünk def utasítássel, akkor megadhatod, hogy mi legyen az érték a return utasítássel.

2 részből áll:

* A return kulcsszó
* Az érték vagy kifejezés (expression) amit vissza kell adnia.

Amikor egy kifejezést használunk egy rturn kifejezéssel, akkor a return érték az az, amit ez a kifeejezés jelent. Például ez az alábbi kód meghatároz egy függvényt, ami visszaad (return) egy string-et a szám alapján amit kap.

import random  
  
def get\_ans(ansnumb):  
  
 if ansnumb == 1 :  
 return 'Ez biztos'  
  
 if ansnumb == 2 :  
 return 'Határozottan így van'  
  
 if ansnumb == 3 :  
 return 'Igen'  
  
 if ansnumb == 4 :  
 return 'A válasz homályos, kérdezz újra'  
  
 if ansnumb == 5 :  
 return 'Kérdezz később'  
  
 if ansnumb == 6 :  
 return 'Koncentrálj és kérdezz újra'  
  
 if ansnumb == 7 :  
 return 'A válaszom nem'  
  
 if ansnumb == 8 :  
 return 'Nem túl jók a kilátások'  
  
 if ansnumb == 9 :  
 return 'Nagyon kétséges'  
  
r = random.randint(1, 9)  
fortune = get\_ans(r)  
print(fortune)

Először import-áljuk a random modult, aztán definiáljuk a get\_ans() függvényt. Mivel definiáljuk, nem hívjuk ezért a program átugrik rajta. Aztán a random.randint() függvényt hívjuk, két argumentumel: 1 és 9. E két szám között egy random interger-t válasz és eltárolja az r nevű változóban.

A get\_ans() függvényt hívja a kód az r arguement-el, a program a get\_ans() függvény elejére ugrik és az r érték az ansnumb paraméterben tárolódik, a függvény kiadja az egyik lehetséges string értéket. A program végrehajtása visszaugrik az a sor mögé, ami hívta a get\_ans() függvényt és a kiadott string-et hozzárendeli a fortune változóhoz, ami aztán print-elődik.

Az utolsó három sort letudtuk volna rövidíteni a következő módon:

print(get\_ans(random.randint(1,9)))

A None érték

A python-ban van egy None nevű érték, ami egy érték hiányát jelenti. A None érték az egyetlen NoneType Data típus. Mint a Boolean True és False értéket, a None-t is nagybetűvel kell kezdeni.

Ez az érték nélküli érték hasznos, ha tárolnod kell valamit, amit nem akarsz összekeverni egy igazi értékkel vagy változóval. Egy hely ahol a None-t használják az a return értéke a print() függvénynek.

A print() függvény kinyomtat szöveget a képernyőre, de nem kell semmilyen értéket visszaadnia, mint a len()- vagy input()-nak kell. De mivel minden függvényhívásnak kell egy return érték, a print() a None-t adja vissza.

>>> **spam = print('Hello!')**  
Hello!  
>>> **None == spam**  
True

A háttérban, Python return None-t rak minden függvény definíció mögé, aminek nincs return statement-je (állítása). Ha a return állítást használod érték nélkül (csak a kulcsszó magában), akkor None-t fogsz visszakapni.

Keyword (kulcsszó) arguemnets (argumentumek) és a print() függvény

A legtöbb argumentumet fel lehet ismeri a pozíciójukról a függvényhívásban. Például a random.randint(1, 10) az más, mint a random.randint(10, 1). Az első egy random intergert fog visszaadni 1 és 10 között, míg a második errort ad.

A kulcsszó argumentumek azonban nem a pozíciójukon keresztül azonosíthatók, hanem a kulcsszó szerint, amit eléjük rakunk a függvényhívásban. A kulcszó argumentumeket gyakran optional (opcionális) paramétereknek használják Például a print() függvénynek ilyenek az end és a sep, hogy specifikáljuk, hogy mit nyomtassunk egy argumentum végén és az argumentumei közben.

Ha a következő programot futtatjuk:

print('Hello')  
print('World')

Az így fog kinézni:

Hello

World

A két string külön, külön soron lesz, mert a print() függvény automatikusan ad egy newline karaktert minden string után, amit kap.

Ha viszont megadod neki az end kulcsszó argumentumet, hogy változtassa meg a newline karaktert egy másik string-re az így fog kinézni:

print('Hello', end='')  
print('World')

És a következőt fogja kiadni:

HelloWorld

A nyomtatot szöveget egy sorban rakja, mert nincs newline karakter a ’Hello’ után. Ehelyett egy üres string van. Ez hasznos, ha ki akarod venni a newline karaktert, amit a program minden print() függvényhívás mögé rak

Hasonlóan, ha több string értéket adsz a print()-nek, a függvény automatikusan szeparálja őket egy space-el.

>>> **print('cats', 'dogs', 'mice')**  
cats dogs mice

De az alap szeparáló string-et is megváltoztathatod a sep kulcsszó használatával:

>>> **print('cats', 'dogs', 'mice', sep=',')**  
cats,dogs,mice

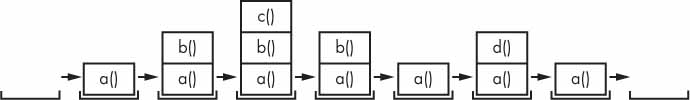
A Call Stack (Hívási verem)

A Python emlékszik hogy melyik sor hívta a függvényt hogy a végrehajtás vissza tudjon oda lépni ha elér egy return állítást, de ha a függvény más függvényeket is hívott akkor először a végrehajtás azokat return-öli az első függvény elött.

def a():  
 print('a() starts')  
 b()  
 d()  
 print('a() returns')  
  
def b():  
 print('b() starts')  
 c()  
 print('b() returns')  
  
def c():  
 print('c() starts')  
 print('c() return')  
  
def d():  
 print('d() starts')  
 print('d() returns')  
  
a()

A program futtatása így néz ki:

a() starts

b() starts

c() starts

c() return

b() returns

d() starts

d() returnsz

a() returns

Amikor az a() függvényt hívjuk, az hívja a b()-t ami szintúgy hívja a c()-t, a c() nem hív semmit csak print()-eli azt hogy, c() starts és c() returns, mielőtt visszamegy ahhoz a sorhoz ami a b()-ben hívta. Aztán a végrehajtás visszalép arra a sorra, ami hívta a b()-t az a()-ban, az ezután tovább megy a program és a végrehajtás hívja a d()-t. Mint a c() függvény a d() nem hív semmit csak print()-el. Ezután visszalép a sorba, ami hívta az a()-ban és a print()-eli hogy a() returns.

A hívási verem az az ahogy a kód emlékszik hogy hova lépjen vissza egy függvényhívás után.

Local and Global Scope (Helyi és globális hatókör)

Paraméterek és változók, amiket egy hívott függvényben adunk meg a függvény helyi hatókörében léteznek (helyi változó). Változók, amiket függvényeken kívül adunk meg a globális hatókörben léteznek (globális változó). Egy változó csak az egyik vagy a másik lehet, nem lehet egyszerre mind a kettő.

Amikor egy hatókör megszűnik a benne lévő változók is elfelejtődnek. Csak egy globális hatókör van, ami akkor jön létre, amikor a program létre jön, amikor a program terminál a globális hatókör megszűnik és a benne lévő változók is elfelejtődnek.

A helyi hatókör akkor jön létre, amikor egy függvényt hívunk. Az összes változó, amit megadunk, a függvényben az csak a függvény helyi hatókörében létezik. Amikor a függvény visszalép a a híváshoz a helyi hatókör megszűnik és a benne lévő változók elfelejtődnek. Legközelebb, amikor hívod a függvényt a helyi változók értékére nem fog emlékezni a programaz előző hívásból.

Hatókörök szabályai:

* A globális hatókörben nem lehet helyi változókat használni
* A helyi hatókörben lehet globális változókat használni
* Egy helyi hatókör nem használhat más helyi hatókörök változóit
* Használhatod ugyanazt a nevet változóknak más-más hatókörökben

A globális hatókörben nem lehet helyi változókat használni

def spam():  
 eggs = 31337  
  
spam()  
print(eggs)

Traceback (most recent call last):

File "C:\pzthon\auto\_bor\c3\loc\_var\_error.py", line 5, in <module>

print(eggs)

^^^^

NameError: name 'eggs' is not defined

Itt amiatt lesz error, mert az eggs változó csak a helyi hatókörben létezik, amikor a spam() függvényt hívjuk. Miután a végrehajtás kilép a spam()-böl a helyi változó elfelejtődik.

Egy helyi hatókör nem használhat más helyi hatókörök változóit

Egy függvény hívásakor új helyi hatókör jön létre, beleértve azt is, ha egy függvényt egy másik függvényben hívnak.

def spam():  
 eggs = 99  
 bacon()  
 print(eggs)  
  
def bacon():  
 ham = 101  
 eggs = 0  
  
spam()

Amikor a program elindul a spam() függvényt hívja a végrehajtás és egy helyi hatókört hoz létre. Az eggs helyi változó értéke 99-re van állítva. Aztán a bacon() függvényt hívja és egy második helyi változó jön létre, több helyi hatókör létezhet egyszerre. Ebben az új helyi hatókörben a ham helyi változó értéke 101 és az eggs helyi változó, ami más változó mint az előző, értéke 0.

Amikor a bacon() visszalép, a helyi értéke megszűnik az eggs változójával együtt.

A helyi hatókörben lehet globális változókat használni

def spam():  
 print(eggs)  
  
eggs = 42  
  
spam()  
  
print(eggs)

Mivel nincs eggs nevű paraméter vagy bármi kód ami ad egy értéket az eggs-nek a spam() függvényben, amikor az eggs használva van a spam-ben, a kód azt a globális változó eggs-re való hivatkozásnak érti. Eza az oka annak hogy a 42 kerül kinyomtatásra.

Használhatod ugyanazt a nevet változóknak más-más hatókörökben

def spam():  
 eggs = 'spam local'  
 print(eggs)  
  
def bacon():  
 eggs = 'bacon local'  
 print(eggs)  
 spam()  
 print(eggs)  
  
eggs = 'global'  
bacon()  
print(eggs)

bacon local

spam local

bacon local

global

Ebben a programban három külön változó van az eggs névvel:

* Egy eggs nevű változó a helyi hatókörben, amikor a spam() függvényt hívja a végrehajtás
* Egy eggs nevű változó a helyi hatókörben, amikor a bacon() függvényt hívja a végrehajtás
* Egy eggs nevű változó a globális hatókörben

A global utasítás

Ha egy globális változót kell változtatnod egy funkcióban, használd a global utasítást

def spam():  
 global eggs  
 eggs = 'spam'  
  
eggs = 'global'  
spam()  
print(eggs)

spam

Mivel az eggs változónál használjuk a global utasítást a spam() függvényben, amikor az eggs értéke ’spam’ lesz ezt a globális hatókörben lévő eggs-en csinálja.

4 szabály van annak a megállapítására, hogy egy változó helyi vagy globális hatókörbe tartozik-e:

* ha a változó a globális hatókörben van használva, akkor az mindig globális változó
* ha van egy global utasítása egy változónak, akkor az egy globális változó
* ha egy változót egy hozzárendelési utasításban használuk egy függvényben, akkor az egy helyi változó.
* ha a változó nincs hozzárendelési utasításban használva, akkor az egy globális változó

def spam():  
 global eggs  
 eggs = 'spam'  
  
def bacon():  
 eggs = 'bacon'  
  
def ham():  
 print(eggs)  
  
eggs = 42  
spam()  
print(eggs)

spam

A spam() függvényben eggs az a globális eggs változó a global utasítás miatt. A bacon() függvényben viszont eggs az egy helyi változó, mert van neki egy hozzárendelési utasítása a függvényben. A ham() függvényben eggs az a globális eggs változó mert nincsen hozzárendelési utasítása (global utasítás sincs).

Egy függvényben a változó az mindig vagy globális vagy helyi lesz.

Ha egy függvényt akarsz, használni egy helyi változón mielőtt adsz neki egy értéket, akkor errort kapsz:

def spam():  
 print(eggs) #ERROR  
 eggs = 'spam local'  
  
eggs = 'global'  
spam()

print(eggs) #ERROR

^^^^

UnboundLocalError: cannot access local variable 'eggs' where it is not associated with a value

Ez amiatt történik, mert Python látja, hogy van egy hozzárendelési utasítás az eggs-nek a spam() függvényben és emiatt az gondolja, hogy eggs helyi változó. De, mivel print(eggs) előbb fut, mint hogy az eggs kap valami értéket a helyi eggs változó nem létezik.

Kivételkezelés (exception handling)

Most, ha egy error-t vagy egy kivételt kapsz az egész program crash-elni fog. Ezt igazi életben nem akarod, hogy megtörténjen, azt akarod, hogy a program észleljen error-okat és kezelje őket

def spam(divide\_by):  
 return 42 / divide\_by  
  
print(spam(2))  
print(spam(12))  
print(spam(0))  
print(spam(1))

21.0

3.5

Traceback (most recent call last):

File "C:\pzthon\auto\_bor\c3\zero\_div.py", line 6, in <module>

print(spam(0))

^^^^^^^

File "C:\pzthon\auto\_bor\c3\zero\_div.py", line 2, in spam

return 42 / divide\_by

~~~^~~~~~~~~~~

ZeroDivisionError: division by zero

Egy ZeroDivisionError történik, amikor nullával próbálsz osztani

Error-okat kezelni lehet a try és az except utasításokkal. A kódba, amiben lehetséges egy error, egy try kifejezést rakunk. Ha error történik akkor a végrehajtás az except kifejezésbe lép ami követi.

def spam(divide\_by):  
 try:  
 return 42 / divide\_by  
  
 except ZeroDivisionError:  
 print('Error: Érvénytelen argumentum')  
  
print(spam(2))  
print(spam(12))  
print(spam(0))  
print(spam(1))

21.0

3.5

Error: Érvénytelen argumentum

None

42.0  
Ne feledje, hogy a try blokk függvényhívásaiban előforduló hibákat is elkapja

def spam(divide\_by):  
 return 42 / divide\_by  
  
try:  
 print(spam(2))  
 print(spam(12))  
 print(spam(0))  
 print(spam(1))  
  
except ZeroDivisionError:  
 print('Error: Érvénytelen argumentum')

21.0

3.5

Error: Érvénytelen argumentum

A print (spam(1)) azért nem fut soha, mert amikor a végrehajtás az except kifejezésbe lép, nem megy vissza a try kifejezésbe többé.

Egy rövid program: Zigzag