4. Zanka while

Zanke pri programiranju uporabljamo, kadar moramo stavek ali skupino stavkov izvršiti večkrat zaporedoma. Namesto, da iste (ali podobne) stavke pišemo *n*-krat, jih napišemo samo enkrat in postavimo v zanko, ki se izvrši *n*-krat. Včasih ne vemo vnaprej, kakšna je vrednost *n*-ja. Zato v splošnem izvajamo zanko dokler je izpolnjen pogoj ponavljanja. Ko pa ta pogoj ni več izpolnjen, se program nadaljuje pri prvem stavku po zanki.
Zanko while v programu napišemo takole:

```
while pogoj:
    stavek1
    stavek2
    stavek3
```

To preberemo takole: dokler je izpolnjen `pogoj`, ponavljaj stavke.
Podobno kot pri pogojnem stavku tudi pri zanki while velja, da lahko en sam stavek v zanki zapišemo takoj za dvopičjem v prvi vrstici.
Pogoj se preveri pred vsako ponovitvijo zanke. Če je izpolnjen, se izvedejo stavki v telesu zanke, sicer pa se izvajanje zanke prekine in se izvede naslednji stavek, ki sledi zanki.
Pri sestavljanju zank moramo paziti, da ne sestavimo pogoja, ki bi bil vedno izpolnjen. Običajno je tako, da moramo v zanki spremeniti vrednost vsaj ene od spremenljivk, ki nastopajo v pogoju. V posebnih primerih imamo tudi zanke, kjer je pogoj vedno izpolnjen, a v takih primerih moramo poskrbeti za drugačen izhod iz zanke (o tem kasneje).
Nalogo, da želimo zanko izvesti n-krat, lahko rešimo takole:

```
n = 20 # število ponavljanj zanke.
i = 0 # števec, ki ga pred zanko inicializiramo
while i < n: #pogoj preverja, če se mora zanka še enkrat
izvesti
      i += 1 # števec vsakih povečamo za eno
      ... # izvajamo stavke v zanki (ki ne spreminjajo
      vrednosti i-ja).
      ... # izvajamo stavke v zanki (ki ne spreminjajo
      vrednosti i-ja).
A # prvi stavek, ki se izvede po koncu zanke.
```

4.1. Zgled: Napišite del programa, ki tabelira kvadrate in kube prvih 20 naravih števil.

```
n = 20 # izpisujemo n vrstic.
i = 0
while i < n:
      i += 1
      print(i,i**2,i**3)
```

4.2. Zgled: Računanje kvadratnega korena pozitivnega realnega števila a.

Uporabimo iteracjo:

$x_{n+1} = (x_n + a/x_n)/2$

Za začetni približek vzamemo:

$x_0 = a$. Uporabimo tudi konstanto ε, ki označuje natančnost računanja.

Rešitev:

```
print("Kvadratni koren iz a z natančnostjo eps. ")
a = float(input("Vtipkaj a: ")
eps = float(input("Vtipaj natančnost eps: ")
      x = 0
      y = (x + a/x)/2
      while abs(y-x) > eps:
            x = y
            y = (x + a/x)/2
print("sqrt( ",a,") =", y)
```

Če bi poznali funkcije, bi bila rešitev lahko takale.

```
def kvadratnikoren(a, eps=0.0000001):
"""kvadratnikoren(a) izracuna kvadratni koren stevil
a."""
      x = 0
      y = (x + a/x)/2
      while abs(y-x) > eps:
            x = y
            y = (x + a/x)/2
      return y
```

4.3. Zgled.

Število pi lahko izračunamo z naslednjo vrsto:

$$\pi = 2\left(1 + \frac{1}{3} + \frac{1\cdot 2}{3\cdot 5} + \frac{1\cdot 2\cdot 3}{3\cdot 5\cdot 7} + \frac{1\cdot 2\cdot 3\cdot 4}{3\cdot 5\cdot 7\cdot 9} + \frac{1\cdot 2\cdot 3\cdot 4\cdot 5}{3\cdot 5\cdot 7\cdot 9\cdot 11} + \cdots\right)$$

Za računanje lahko uporabimo naslednji del programa:

```
#!/usr/bin/python3
# računanje števila pi
pi2 = 0
f = 1
n = 0
m = 1
while f >0.00000001:
    pi2 +=f
    print(2*pi2,n,m,f)
    n += 1
    m += 2
    f *= n/m
print("pi = ",2*pi2)
from math import pi
print("pi = ",pi)
print("razlika = ", 2*pi2-pi)
```

4.4. Naloge:

1. Primerjajte rezultat te funkcije z rezultatom, ki ga dobite z uporabo modula math iz standardne knjižnice.
2. Popravite program tako, da izpisuje v vrstico korak iteracije n in približek v n-ti iteraciji.
3. Spremenite vrednost parametra `eps`, tako da povečate natančnost rezultata na 200 decimalnih mest.
4. Izračunajte kvadratni koren prvih 10 naravnih števil.
5. Sestavite funkcijo izpisiNaravnaDon(n), ki izpiše vsa naravna števila od 1 do *n*.
6. Sestavite funkcijo vsotaStevk(n), ki vrne vsoto števk celega števila *n*.
7. Sestavite funkcijo izpisiFibonacciDon(n,a=1,b=1), ki izpiše prvih *n* členov posplošenega fibonaccijevega zaporedja (prvi in drugi člen dobi za parametra, vsak naslednji člen je enak vsoti prejšnjih dveh).
8. Sestavite funkcijo jePrastevilo(n), ki preveri, ali je naravno število *n* praštevilo.
9. Sestavite funkcijo multiFakulteta(n, s), ki izračuna multifakulteto.

10. Sestavite funkcijo exp(x), ki izračuna z vrsto $1 + x + x^2/2! + x^3/3! + \ldots$ vrednost funkcije e^x. Nato izračunajte konstanto e na 6 decimalnih mest.

11. Izračunajte število pi na 6 decimalnih mest. Namig: lahko uporabite metodo, s katero je število pi računal veliki slovenski matematik Jurij Vega.

4.5. Stavki break, continue, pass, povezani z zankami

Delo z zankami si lahko včasih poenostavimo, če uporabljamo posebne stavke:

- `break`
- `continue`
- `pass`

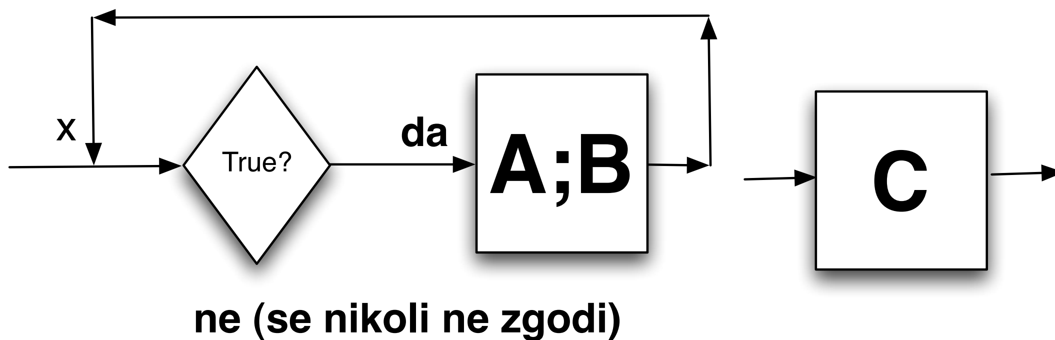
`break` - izhod iz najožje zanke
`continue` - naslednji prehod zanke
`pass` - ne naredi ničesar.

Ti stavki niso nujno potrebni. Lahko programiramo tudi brez njih. Včasih pa so kar koristni. Predvsem stavek `break`.

4.6. Stavek `break`,

Včasih si s stavkom `break` pomagamo pri "neskončnih" zankah. Zanko, ki se izvaja v neskončnost, dosežemo npr. takole:

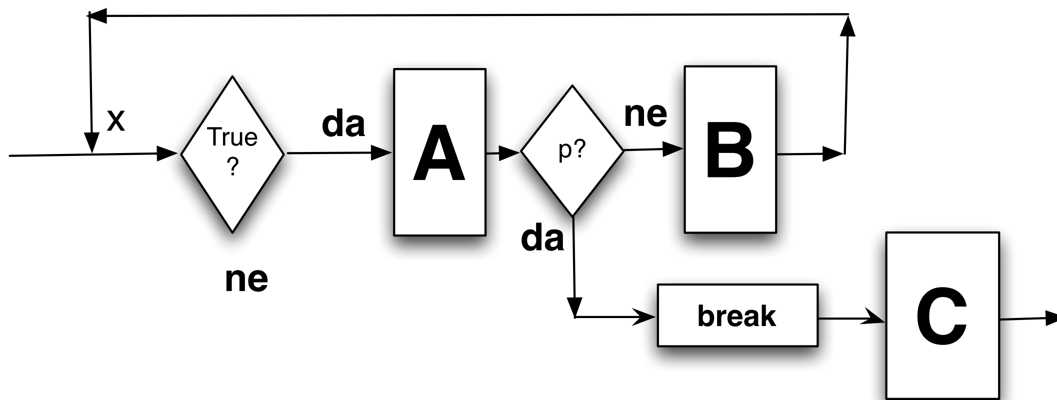
```
while True:
   A
   B
C
```



Stavka A in B se izvajata *ad infinitum*. Zanka se nikoli ne konča. Stavek C se ne izvede nikoli.

Stavek `break` lahko prekine tek neskončne zanke. Npr. takole

```
while True:
   A
   if p:
      break
   B
C
```



Če imamo težave pri oblikovanju pogoja za izhod iz zanke, je najbolje uporabiti neskončno zanko (`True`) in pogoj `p` oblikovati v kombinaciji s stavkom `break` takrat, ko ga potrebujemo, da bi zapustili zanko.

4.6. Zgled: Za dano število n ugotovite, ali je deljivo s kakšnim praštevilom oblike p = 6k + 1.

```
n = 2012
p = 2
odg = False
while True:
   if n%p == 0 and p%6 == 1:
      odg = True
      break
   while n%p == 0:
      n //= p
      if n == 1:
         break
      p += 1
print(odg)
```

4.7. Stavek `continue`.

S stavkom `continue` prekinemo trenutno delo in nadaljujemo z naslednjim prehodom zanke.

```
while True:
   A
   if p:
      continue
   B
C
```

4.8. Stavek `pass`.

Stavek `pass` ne naredi nič. Uporabljamo ga predvsem kot mašilo, da zadostimo pravilom sintakse jezika. Najbolj običajno mesto zanj je takoj za `if` in pred `else` ali `elif`.

```
if a > 0:
    pass
else:
    a = - a
```

4.9. Razširjeni stavek while

Stavek `while`, v katerem je mogoč izhod z ukazom `break` lahko razširimo takole:

```
while p:
   A
   if q:
      C
      break
else:
   B
D
```

V zgornjem primeru se stavek D vedno izvede.
Stavek B pa se izvede po običajnem izhodu iz zanke, če ni izhod izsiljen s stavkom `break`.

4.7. Naloga(*): Poiščite problem, pri kateri uporaba razširjenega stavka while poenostavi programiranje.

4.8. Zgled: Colatzovo zaporedje. Začetni člen je poljubno celo število:
$x_0 = A$.

Uporabimo iteracjo:
če je število x_n sodo je

$x_{n+1} = x_n/2$

sicer pa je

$x_{n+1} = 3x_n + 1$.

Zapište program, ki izpisuje člene tega zaporedja, dokler ne naleti na vrednost 1.

```
# Colatz
x = int(input("Vpiši število : "))
while x > 1:
    print(x)
    if x%2 == 0:
        x //= 2
    else:
        x = 3*x + 1
print(x)
```

4.9. Zgled: Iskanje maksimuma funkcije f(x) na zaprtem intervalu [a,b].

Naslednji del programa poišče lokalni maksimum funkcije f(x). V našem primeru je f(x) = x**2, zato je ekstrem globalen.

```
def f(x):
   return -x**2

eps = 0.00001
a = float(input(" Levo krajišče intervala a: "))
b = float(input(" Desno krajišče intervala a: "))
c = (a+b)/2
if f(a) <= f(c) <= f(b):
   while abs(a-b) > eps and f(c) <= f(b):
      a = c
      c = (a+b)/2
elif f(a) >= f(c) >= f(b):
   while abs(a-b) > eps and f(a) >= f(c):
      b = c
      c = (a+b)/2
# pogoj je f(c) >= max(f(a), f(b))
while abs(a-b) > eps:
   c = (a + b)/2
   d = (a+c)/2
   e = (b+c)/2
   if f(d) >= f(a) and f(d) >= f(c):
      b = c
   elif f(e) >= f(c) and f(e) >= f(b):
      a = c
   else:
      a = d
      b = e
cc = int(c*100000)/100000
fcc = int(f(c)*100000)/100000
print(cc, fcc)
```

Ker vsebuje ta program skrito napako, smo ga še enkrat sprogramirali.

```
def f(x):
   return -x**2

while True:
   a = float(input("a = "))
   b = float(input("b = "))
   if b < a:
      print("The End")
      break
   c = (a + b)/2
   if f(a) >= f(c):
      while f(a) >= f(c) and abs(a-b) > 0.000001:
         b = c
         c = (a + b)/2
   elif f(b) >= f(c):
      while f(b) >= f(c) and abs(a-b) > 0.000001:
         a = c
         c = (a + b)/2
   while abs(a-b) > 0.000001:
      c = (a + b)/2
      d = (a + c)/2
      e = (c + b)/2
      if f(d) >= f(c):
         b = c
      elif f(e) >= f(c):
         a = c
      else:
         a = d
         b = e
   c = (a + b)/2
   fc = f(c)
   cc = int(c * 10**6)/10**6
   fcc = int(fc*10**6)/10**6
   print("Maksimum je ", fcc," in ga dosežemo pri ",cc)
```

Za vajo lahko poiščete napako v prvem programu, tako da ga primerjate z drugim programom. Poskusite odgovoriti na naslednja vprašanja:

1. Drugi program ima eno zanko več kot prvi. Čemu je namenjena in kako jo med tekom programa zapustimo?
2. Čemu smo vpeljali spremenljivki cc in fcc. Zakaj nismo izpisali kar vrednosti c in f(c)?
3. Poiščite vrednosti za a in b pri katreih programa dajeta različne odgovore. Pojasnite to razliko.
4. Napaka v prvem programu je latentna in je pri funkciji f(x) = - x**2 ne opazimo. Izberite drugačno funkcijo f(x) in taki krajišči intervalov a in b, da bo prvi program izračunal napačno vrednost lokalnega maksimuma funkcije f.

4.10. Univerzalna moč računanja.
Z doslej spoznanimi programskimi konstrukti lahko zdaj zapišemo poljubno kompliciran program. Pri podatkih pa imamo sicer na voljo poljubno dolge nize, za praktično računanje pa bi potrebovali še sezname. Do njih bomo prišli prihodnjič.

Python pa omogoča tudi nekatere konstrukte, nekatere bomo spoznali kasneje, ki niso popolnoma v skladu s paradigmo strukturiranega programiranja, vendar nam včasih poenostavijo tako razmišljanje, kakor tudi programiranje.