10. Standardna knjižnica - Moduli - Import

Python pride skupaj s številnimi moduli oz. paketi, ki ob zagonu tolmača sicer niso neposredno dostopni, lahko pa jih naložimo, če jih potrebujemo. Tej zbirki modulaov pravimo standarsna knjižnica. V standardni knjižnici je prek 100 modulov. Običajno pa ne potrebujemo več kot peščico. Na ta način sta zagon tolmača pa tudi nalaganje programa hitrejša.
Funkcije danega modula moramo najprej vključiti na enega od spodnjih načinov:

- `from MODUL import *`
- `import MODUL`

Pri drugem načinu, ki je bolj varen, moramo pred ime vsake nove funkcije dodati `MODUL`, npr. `MODUL.f(x)`.
Pri prvem načinu to ni potrebno. Če uporabimo zvezdico, dobimo vse funkcije danega modula. Bolj varno je, če naložimo le tiste funkcije, ki jih res želimo, npr.

- `from MODUL import f`

Težava utegne nastopiti, če tudi sami definiramo funkcijo z istim imenom f. Katera velja? Če uporabimo zvezdico, zgubimo pregled nad tem, katera imena smo naložili in prav lahko se zgodi, da pride do konflikta.

Pakete, oz. module lahko dobimo tudi iz drugih, neuradnih virov ali pa jih pišemo tudi sami, vendar se te veščine ne bomo učili pri uvodnem predmetu.

Včasih damo modulu tudi svoje, običajno krajše ime. Npr. namesto

- `import time`

zapišemo

- `import time as t`

10.1. Matematične funkcije (modul math)

Matematične funkcije moramo najprej vključiti na enega izmed spodnjih načinov:

- `from math import *`
- `import math`

Pri drugem načinu moramo pred ime vsake matematične funkcije dodati math., npr. math.sin(x). Pri prvem načinu to ni potrebno.

- sqrt(x) - kvadratni koren
- sin(x), cos(x), tan(x) - trigonometrične funkcije
- exp(x), log(x) - eksponentna in logaritemska funkcija
- pi, e - konstanti 3.14 in 2.71

1. `math.ceil(`*x*`)`
2. `math.copysign(`*x*`, `*y*`)`
3. `math.fabs(`*x*`)`
4. `math.factorial(`*x*`)`
5. `math.floor(`*x*`)`
6. `math.fmod(`*x*`, `*y*`)`
7. `math.frexp(`*x*`)`
8. `math.fsum(`*iterable*`)`
9. `math.isinf(`*x*`)`
10. `math.isnan(`*x*`)`
11. `math.ldexp(`*x*`, `*i*`)`
12. `math.modf(`*x*`)`
13. `math.trunc(`*x*`)`
14. `math.exp(`*x*`)`
15. `math.log(`*x*`[, `*base*`])`
16. `math.log1p(`*x*`)`

17. `math.log10(`x`)`
18. `math.pow(`x, y`)`
19. `math.sqrt(`x`)`
20. `math.acos(`x`)`
21. `math.asin(`x`)`
22. `math.atan(`x`)`
23. `math.atan2(`y, x`)`
24. `math.cos(`x`)`
25. `math.hypot(`x, y`)`
26. `math.sin(`x`)`
27. `math.tan(`x`)`
28. `math.degrees(`x`)`
29. `math.radians(`x`)`
30. `math.acosh(`x`)`
31. `math.asinh(`x`)`
32. `math.atanh(`x`)`
33. `math.cosh(`x`)`
34. `math.sinh(`x`)`
35. `math.tanh(`x`)`
36. `math.pi`
37. `math.e`

10.2. Sorodni moduli.

Za opis seznama matematičnih funkcij glej dokumentacijo. Ob modulu math obstajajo še moduli

- `cmath`
- `number`
- `fraction`
- `decimal`

10.3. Naloge:

1. Napišite program, ki reši transcendentno enačbo: x = cos(x).
2. Napišite funkcijo, ki tabelira logaritemsko funkcijo na danem intervalu (a,b), z danim korakom d.

10.4. Modul random

Običajno računanje je deterministično. To med drugim pomeni, da je izračun ponovljiv. Pri enakih podatkih dobimo enake rezultate. Na voljo pa imamo tudi paket, ki računanje naredi verjetnostno. V paketu **`random`** imamo na razpolago funkcije za generiranje naključnih števil:

- **`choice(s)`** ... naključni element iz seznama s
- **`random()`** ... naključno realno število z intervala [0, 1)
- **`randint(a, b)`** ... naključno celo število z intervala [a, b]
- **`randrange(a, b, k)`** ... naključno celo število iz **`range(a, b, k)`**
- **`shuffle(s)`** ... naključna permutacija seznama s.

10.5. Naloge:

1. Simulacija metanja kocke pri igri "Človek ne jezi se". Napišite funkcijo koraki(), ki pove za koliko korakov se premaknemo, po metu kocke. Pri tem upoštevamo, da po vsaki šestici še enkrat mečemo kocko.

```
from random import randint

def koraki():
  k = 0
  while True:
    k0 = randint(1,6)
    k + = k0
    if k0 < 6:
    break
  return k
```

2. Sporočilo je dolgo 100 bitov. Pošljemo ga n krat. Sporočilo moti šum, ki je enakomerno porazdeljen med 0 in 10 (je lahko 10 krat močnejši od sporočila). Sprejeta sporočila z napakami sešetejemo in poskušamo razpoznati poslano sporočilo. Izvedite simulacijo.
3. Simulacija igre nim(n,k). Na začteku računalnik naključno izbere velikost kupčka n, največje število k kamnov, ki jih je dovoljeno vzeti iz kupčka ter vpraša igralca, ali bi rad začel. Potem izmenično jemljeta kamne iz kupčka. Zmaga tisti, ki vzame zadnji kamen in sprazni kup. Računalnik na koncu razglasi zmagovalca.
4. Napišite program, ki naključno izbere in deli talon (šest kart) pri taroku.
5. Napišite program deli(k), ki meša in deli karte pri taroku za tri (k=3) ali za štiri (k=4) igralce.
6. Sestavite funkcijo, ki simulira met koce pri igri človek ne jezi se.
7. Sestavite funkcijo, ki meša in deli karte za bridge.
8. Sestavite funkcijo, ki določi frekvence znakov (črk) v danem beedilu. Na osnovi dobljenih frekvenc naključno generirajte novo besedilo.
9. Ponovite prejšnjo nalogo, samo da namesto frekvenc znakov naračunate in pri generiranju uporabite frekvence parov znakov.

10.6. Modul time

Modul time pozna več funkcij povezanih s časom in datumom. Če želimo preveriti kako hitro se izvaja kakšen program, uporabljamo funkcijo clock(), ki meri čas centralno procesne enote (CPU time). Z modulom time lahko prrimerjamo hitrost izvajanja dveh programov pri istih podatkih ali pa nam pomaga ugotoviti, kako hitro se povečuje čas računanaja pri obsežnejših podatkih. S tem problemi se ukvarja področje računalništva, ki mu pravimo teorija časovne zahtevnosti.

```
from time import clock
n = 100000
start = clock()
i = 0
while i < n:
     i += 1
stop = clock()
print(stop-start)
```

10.7. Sorodni moduli:

- datetime
- locale
- calendar

10.8. Naloge

1. Koliko časa porabimo, da izračunamo rekurzivno n-to fibonaccijevo število f(n)? Tabelirajte n, f(n) in pripadajoči čas za n do 25. Naloga uporablja metodo format, ki jo bomo spoznali kasneje.

```
from time import clock
def f(n):
	if n < 2:
		return 1
	return f(n-1) + f(n-2)
for n in range(36):
	z = clock()
	odg = f(n)
	k = clock()
	print("{0:5}{1:10}{2:5.2f}".format(n,odg, k-z))
```

2. Poiščite dokumentacijo za standardno knjižnico in preizkusite delovanje vsaj petih modulov.

3. S funkcijo clock preverite ali je brisanje elementa z dolgega seznama časovno bolj potratno kot brisanje elementa z enakodolgega niza.

10.9. Modul turtle

Modul turtle omogoča delo z želvjo grafiko. Želvja grafika je temelj zanimivega nekoč zelo popularnega programskega jezika LOGO. Želva je nekakšna pero, običajno v obliki puščice, ki se premika po zaslonu, oz. po risalni površini in za seboj pušča sled. Modul turtle pozna zelo veliko ukazov za delo z želvami. Uporablili jih bomo ssmo nekaj. Puščica kaže smer, v katero je trenutno usmerjena želva.
Peresu rečemo želva. V vsakem trenutku se nahaja v nekem stanju. Stanje ima več komponent.

- Položaj - koordinati x in y.
- Dvignjeno-spuščeno pero.
- Barva peresa.
- Smer peresa, oz. želve.
- Ostale parametre stanja lahko razberemo iz spodnjih ukazov oz. s poskušanjem.

Za preprosto risbo zadoščata že ukaza fd() in rt():
fd(d) ... naprej za d korakov v trenutni smeri.
rt(a) ... zasuk na desno za a stopinj na trenutnem položaju.

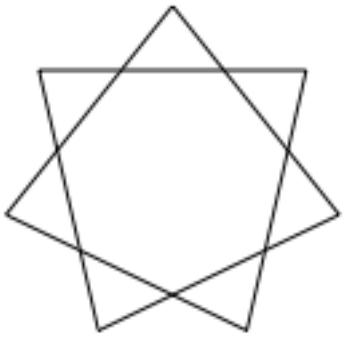
10.10 Zgled:

Narišemo zvezdo:

```
def zvezda(n,r,d=50):
	"""
	Narišemo zvezdo z n kraki in korakom r.
	"""
	for i in range(n):
		fd(d)
		rt(r*360/n)
```

zvezda(5,2) nariše petkrako zvezdo.

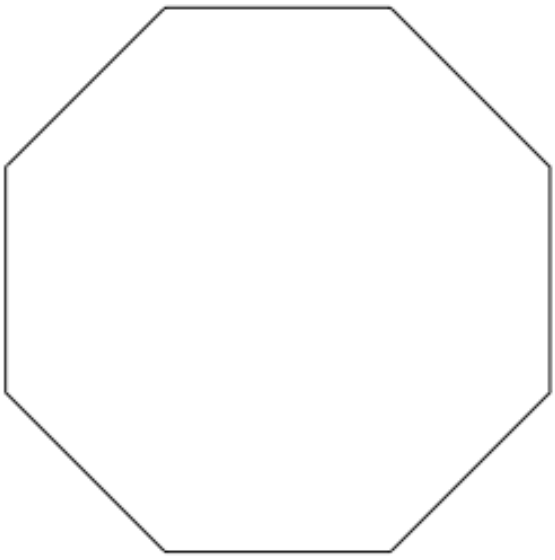
zvezda(7,2)



zvezda(7,3)

zvezda(6,2) pa nariše (napčno!) trikotnik,

zvezda(8,1) pa



osemkotnik

Pojavlja se kar nekaj vprašanj:

- Kako moramo popraviti program, da bo deloval tudi za "nepovezane" zvezde?
- Kako moramo popraviti program, da bodo narisane zvezde enako velike?
- Kako moramo popraviti program, da bo risal zvezde v obliki lika (brez presečišč)?

Pri reševanju teh problemov potrebujemo tudi malo bolj zapletene ukaze.
Nepogrešljivi so:

- **`pd()`** spusti pero
- **`pu()`** dvigni pero
- **`reset()`** pobriši zaslon in se vrni v začetni položaj.

10.11. Pregled metod za želvo (Turtle) in zaslon (Screen).

Povzeto po "uradni" dokumentaciji. S klikom na metodo dobimo pojasnilo v angleščini.

Metode za želvo

10.12. Premik želve

A) Premiki in risanje

```
forward()|fd() : naprej v smeri želve za d enot

backward()|bk()|back() : nazaj za d enot.

right()|rt() : zasuk na mestu v desno za kot a.

left()|lt() : zasuk na mestu v levo za kot a.

goto()|setpos()|setposition() : premik v točko (x,y)

setx() premik v vodoravni smeri v točko (x,?)

sety() premik v navpični smeri v točko (?,y)

setheading()|seth() : zasuk na mestu v smer kota a.

home() : premik v izhodišče

circle() : nariši krog.

dot() : nariši točko

stamp() : pusti sled želve.

clearstamp() : zbriši zadnjo sled želve

clearstamps() : zbriši vse sledi želve

undo() : pozabi zadnji ukaz

speed() : hitrost želve
```

B) Stanje želve

```
position()|pos() : položaj želve (x,y)
towards() : premakni se proti točki (x,y)
xcor() : koordinata x.
ycor() : koordinata y.
heading() : smerni kot a:
distance() : razdalja od želve do točke (x,y)
```

C) Merjenje kotov

```
degrees() : enota za kot je stopinja.
radians() : enota za kto je radian.
```

10.13. Kontrola peresa

A) Risalno stanje

```
pendown()|pd()|down() : spusti pero.
penup()|pu()|up() : dvigni pero.
pensize()|width() : širina peresa.
pen() : pero
isdown() : je pero spuščeno?
```

B) Kontrola barve

```
color() : barva.
pencolor() barva peresa.
fillcolor() : barva ploskve.
```

C) Polnjenje

```
filling() : polnjenje
begin_fill() začni polniti.
end_fill() končaj s polnjenjem.
```

D) Kontrola risanja

```
reset() : zbriši vse in se vrni v izhodiščni položaj.
clear() : zbriši zaslon.
write() : napiši besedilo na ternutni polžaj.
```

10.14. Želvino stanje

A) Vidnost želve

```
showturtle()|st() : pokaži želvo

hideturtle()|ht() : skrij želvo

isvisible() : je želva vidna?
```

B) Oblika želve

```
shape() : nastavi obliko

resizemode() : način povečave

shapesize()|turtlesize() : ??
settiltangle() : ??

tiltangle() : ??

tilt() : ??
```

10.15. Uporaba dogodkov (events)*

```
onclick() : kaj izvedemo ob kliku

onrelease() : kaj se izvede, ko spustimo miš

ondrag() : kaj se izvede ob vleku
```

10.16. Posebne želvine metode

```
begin_poly() : začnemo beležiti poligon.

end_poly() : končamo z beleženjem poligona.

get_poly() : dobimo trenutni poligon.

clone() : kloniramo želvo.

getturtle()|getpen()
getscreen()
setundobuffer()
undobufferentries()
```

10.17. Metode zaslona

A) Kontrola okna

```
bgcolor()
bgpic()
clear()|clearscreen()
reset()|resetscreen()
screensize()
setworldcoordinates()
```

B) Kontrola animacije

```
delay()
tracer()
update()
```

C) Uporaba zaslonskih dogodkov

```
listen()
onkey()
```

```
onclick()|onscreenclick()
ontimer()
```

D) Nastavitve in posebne metode

```
mode()
colormode()
getcanvas()
getshapes()
register_shape()|addshape()
turtles()
window_height()
window_width()
```

E) Metode za zaslon

```
bye() : zapremo zaslon

exitonclick()
setup() : določimo velikost in položaj zaslona.

title() : naslov zaslona.
```

10.18. Kako shranimo sliko, ki jo nariše želva?

Z naslednjimi tremi ukazi zapišemo sliko v eps obliki na datoteko z imenom f in zapremo želvin zaslon.

```
zaslon = getscreen()
zaslon.getcanvas().postscript(file=f)
bye()
```

10.19. Rekurzija in želvja grafika

Z uporabo rekurzije pri želvji grafiki lahko dosežemo zelo zanimive grafične efekte.

10.20. Naloge:

1. Napišite program, ki nariše Kochovo snežinko.
2. Napišite program, ki nariše Hilbertovo krivuljo.
3. Napišite program, ki nariše krivuljo Sierpinskega.