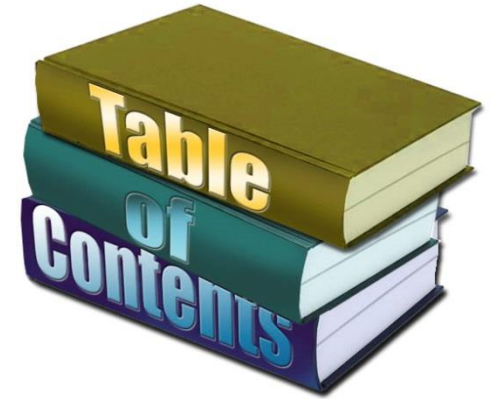


PROGRAMIRANJE

2014/15

*o programskih jezikih
primerjava FP in OUP
FP v Pythonu*

Pregled

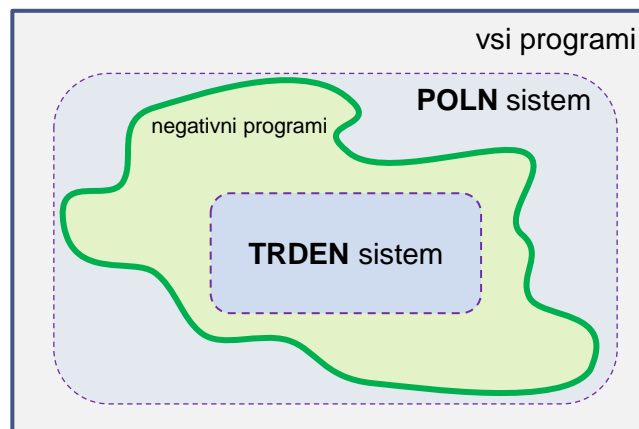


- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja
- funkcijsko programiranje v Pythonu
- zaključek

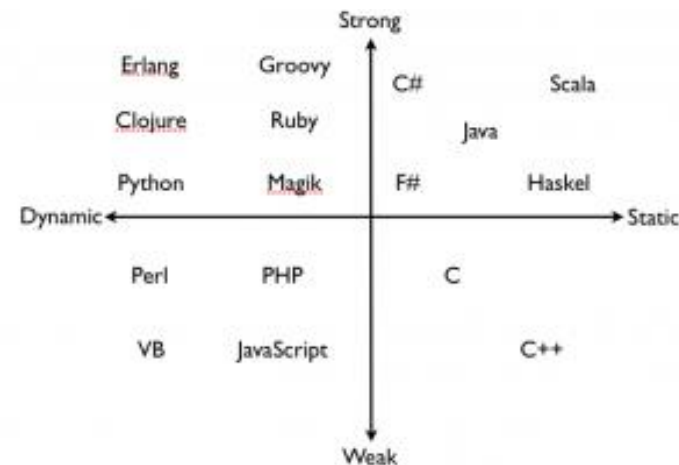
Trdnost in polnost sistema tipov

- terminologija:
 - pozitiven primer** programa: program, ki ima napako (+)
 - negativen primer** programa: program brez napake (-)
- sistem je TRDEN** (angl. *sound*), če nikoli ne sprejme pozitivnega programa
 - statično preverjanje zavrne lažne pozitivne primere (= pravilni programi, ki jih zaznamo kot nepravilne)
- sistem je POLN** (angl. *complete*), če nikoli ne zavrne negativnega programa
 - statično preverjanje sprejme lažne negativne primere (= nepravilni programi, zaznani kot pravilni)

analiziran kot program	P	N
P	PP	LN
N	LP	PN











Šibko tipiziranje



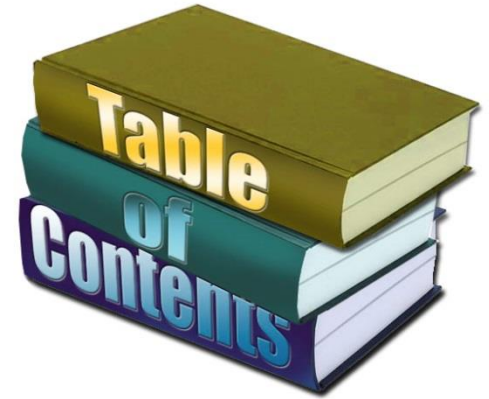
- **šibko tipiziranje** (angl. *weak typing*)
- sistem, ki:
 - NI TRDEN (dopušča lažne negativne primere)
 - med izvajanjem se lahko zgodi napaka (potrebno preverjanje)
 - sistem izvaja zelo malo (premalo) statičnih ali dinamičnih preverjanj
 - rezultat dopustnih programov: neznan?
 - C / C++
- prednosti šibko tipiziranih sistemov
 - "omogočajo večje programersko mojstrstvo?"
 - lažja implementacija programskega jezika (ni avtomatskih preverjanj)
 - večja učinkovitost (čas za preverjanja; prostor za oznake podatkovnih tipov)

Prednosti obeh sistemov

	statično preverjanje	dinamično preverjanje
kombiniranje podatkovnih tipov v seznamih in vejah programa	ni možno, vendar pa zato v programu vemo, katere tipe seznamov in funkcij lahko pričakujemo 	je možno, moramo pa v programu uporabiti vgrajene predikate za preverjanje tipov 
sprejemanje množice programov	zavrača pravilne programe 	sprejema več pravilnih programov 
čas ugotavljanja napak	napake v programu ugotovimo zgodaj 	napake ugotovi šele med izvajanjem 
hitrost izvajanja	prihrani na prostoru in času, ker ne označuje spremenljivk z značkami posameznih podatkovnih tipov 	prevajalnik potrebuje več prostora in časa za označevanje spremenljivk z značkami, programer ima več dela 
večkratna uporabnost programske kode	manjša, vendar s tem večje nadzorovanje napak 	večja, vendar odpira možnosti za več napak v programu 
prototipiranje novih programov	težje, potrebno določiti podatkovne tipe vnaprej; vendar pa lažje nadzorovanje vpliva sprememb na delovanje obstoječe kode 	preprosteje, vendar slabše nadzorovanje vpliva sprememb na obstoječo kodo 

- kaj je boljše?
- smiselno je najti kompromis: del preverjanja se izvede statično, del dinamično
- programski jeziki uporabljajo kombinacijo obojega

Pregled



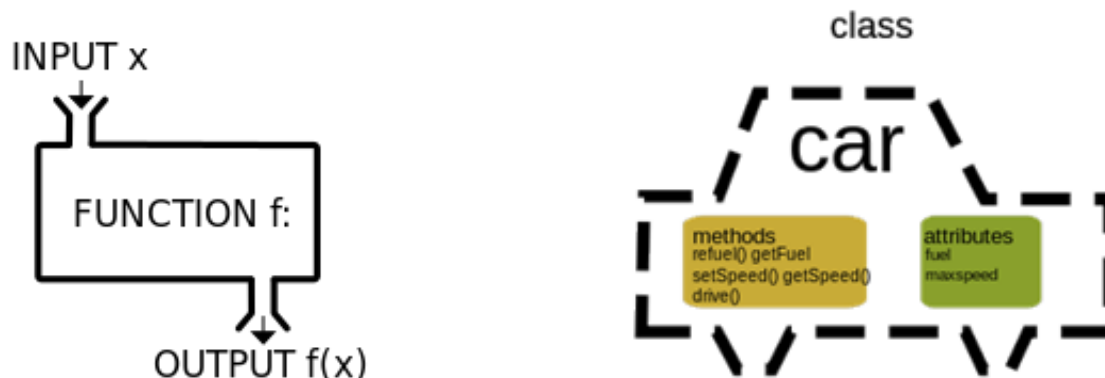
- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja
- funkcijsko programiranje v Pythonu
- zaključek

Povezava med FP in OUP

- program lahko analiziramo glede na uporabo podatkovnih tipov in funkcij:

	funkcija1	funkcija2	funkcija2	...
tip1				
tip2				
tip3				
...				

- "narediti program" pomeni "izpolniti" zgornjo tabelo s programsko kodo za vsak podatkovni tip in funkcijo, ki jo uporabljamo



Povezava med FP in OUP





- **funkcijsko programiranje:** program je množica funkcij, ki so zadolžene vsaka za svojo operacijo

	funkcija1	funkcija2	funkcija2	...
tip1				
tip2				
tip3				
...				

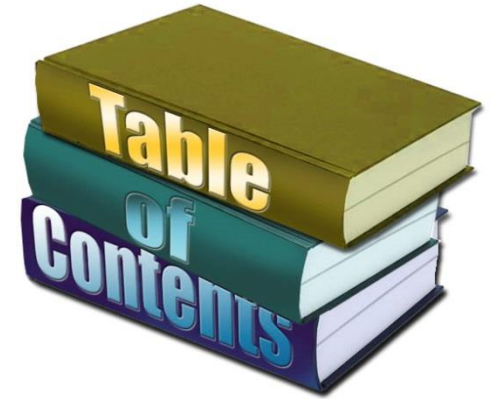
- **objektno-usmerjeno programiranje:** program je množica razredov, ki v sebi vsebujejo različne operacije nad primerki razreda

	funkcija1	funkcija2	funkcija2	...
tip1				
tip2				
tip3				
...				

Povezava med FP in OUP

- FP in OUP sta torej le drugačni perspektivi na izdelavo istih programov
- **katerega izbrati?**
 - osebni programerski stil
 - upoštevati je potrebno način razširjanja programa
- **razširjanje programa z novo kodo:**
 - funkcijsko programiranje
 - če dodamo novo funkcijo, moramo v njej pokriti vse konstruktorje za podatkovni tip (sicer nas prevajalnik opozori) 
 - če samo razširimo podatkovni tip, moramo dopolniti vse funkcije s kodo za delo s tem tipom 
 - objektno-usmerjeno programiranje
 - če dodamo novo funkcijo za delo s podatkovnimi tipi, jo moramo implementirati na novo v vseh ločenih razredih (ne upoštevajmo možnosti dedovanja) 
 - če implementiramo novi razred, v njemu implementiramo vse možne funkcije (metode) za delo s tem razredom 

Pregled



- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja
- funkcijsko programiranje v Pythonu
- zaključek

FP v Pythonu

- Python ni čisti funkcijski jezik, nudi različne paradigme programiranja
- ima zmožnosti funkcijskega programiranja
 - anonimne funkcije
 - funkcije višjega reda
 - `map()`, `reduce()`, `filter()`
 - izpeljani sezname
 - iteratorji
 - generatorji
 - in še druge...
- uporaba anonimnih funkcij

```
def kvadrat(x):  
    return x**2
```

```
kvadrat = lambda x: x**2
```



Funkcije višjega reda

- definicija funkcij višjega reda

```
def izbira(ocena):  
    if ocena > 5:  
        return lambda dan: "V " + dan + " praznujemo."  
    else:  
        return lambda tocke, datum:  
            "Dobil sem samo " + str(tocke)  
            + " tock. Dne " + datum  
            + " je naslednji rok."  
  
>>> rezultat = izbira(5)  
>>> rezultat(33, "1.1.2012")  
'Dobil sem samo 33 tock. Dne 1.1.2012 je naslednji rok.'  
  
>>> rezultat = izbira(10)  
>>> rezultat("petek")  
'V petek praznujemo.'
```



- uporaba vgrajenih funkcij

```
>>> map(lambda x: x**2, range(1,5))  
[1, 4, 9, 16]  
>>> filter(lambda x: x%2==0, range(10))  
[0, 2, 4, 6, 8]  
>>> reduce(lambda x,y: x+y, [47, 11, 42, 13])  
113
```

Funkcijske ovojnice

- ovojnica = telo funkcije + okolje
- možni načini uporabe :
 - uporaba privzetih vrednosti
 - ovoj funkcije v zunanjo funkcijo
 - uporaba posebnih razredov in orodij (closure, Bindings)

dinamični doseg

```
>>> N = 10
>>> def pristejN(i):
    return i+N
```

```
>>> pristejN(7)
17
>>> N = 20
>>> pristejN(7)
27
```

zaprtje s privzeto vrednostjo
("zapečena" v funkcijo)

```
>>> N = 10
>>> def pristejN(i, n=N):
    return i+n
```

```
>>> pristejN(7)
17
>>> N = 20
>>> pristejN(7)
17
```

zaprtje z ovijanjem funkcije
v zunanjo funkcijo

```
>>> N = 10
>>> def pristejNx(N):
    def pristejN1(i):
        return i+N
    return pristejN1
>>> moja = pristejNx(10)
>>> moja(5)
15
>>> N = 20
>>> moja(5)
15
```

Sestavljeni seznam

- način avtomatskega sestavljanja seznamov v skladu z matematično formulacijo
- lahko nadomestijo map, filter in reduce,
- $A = \{f(x) \mid x \in D, \text{pogoj}(x)\}$, npr. $A = \{x^2 \mid x \in \mathbb{N}, 5 \leq x \leq 15\}$
- sestavljen seznam (list comprehension)

```
[funkcija(x) for x in D if pogoj(x)]
```

je enakovredno

```
r = []  
for e in D:  
    if pogoj(e):  
        r.append(funkcija(e))
```

- primer:

```
def prasteviloS(n):  
    return not [x for x in range(2, n) if n % x == 0]
```

Iterator

- objekt, ki predstavlja tok podatkov, vrača posamezne elemente
 - zakasnjeno izvajanje
- implementira metodi:
 - `__iter__()` - vrne objekt-iterator
 - `__next__()` - vrne naslednji element do izčrpanja (**StopIteration**) ali v neskončnost
- premikanje nazaj ni možno
- funkcija **iter(objekt)** - pretvorba objekta v iterator

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def next(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

for c in Counter(3, 8):
    print c
```

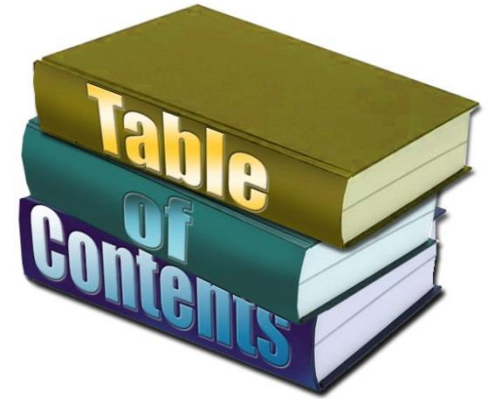
Generator

- generatorji so funkcije, ki iterirajo preko toka vrednosti:
 - omogočajo zakasnjeno izvajanje izrazov (šele, ko jih potrebujemo)
 - izvajanje funkcije se lahko zaustavi in nadaljuje
 - lokalno okolje funkcije se ohranja tudi ob zaustavitvi
 - stavek **yield** zaustavi izvajanje in preda kontrolo klicočemu okolju
 - funkcija nadaljuje z izvajanjem ob klicu metode **__next__()**
 - generiranje se zaključi s stavkom **return** ali izjemo **StopIteration**

```
def genstevec(max):  
    i = 0  
    while i < max:  
        yield i  
        i += 1
```

```
def genstevec(max):  
    i = 0  
    while i < max:  
        vnos = (yield i)  
        if vnos == None:  
            i += 1  
        elif vnos >= max:  
            raise StopIteration  
        else:  
            i = vnos
```


Pregled



- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja
- funkcijsko programiranje v Pythonu
- zaključek

Zaključek

cilji predmeta:

- postati boljši programer
 - naučiti se novih konceptov (polimorfizem, zakasnjena evalvacija, tokovi, memoizacija, funkcije višjega reda, ovojnice, delna aplikacija, currying, ...)
- razumeti delovanje programskega jezika
 - pridobiti sposobnost hitrega učenja novega programskega jezika
 - ločiti bolj in manj elegantne implementacije
- izstopiti iz okvira objektno-usmerjenega programiranja
 - razumeti funkcijsko in objektno-usmerjeno paradigmo
- naučiti se funkcijskega programiranja. Njegove prednosti:
 - bolj abstrakten opis problema
 - možnost paralelizacije
 - brez mutacije vrednosti (manj semantičnih napak)
 - lažje testiranje (testi enot)
 - lažji formalni dokaz pravilnosti
- dojeti, zakaj nekdo, ki napiše [tak članek](#) smeši in diskreditira samega sebe ☺





May the λ be with You!