

Python ne zahteva (in niti ne omogoča) *deklaracije* spremenljivk, temveč le *definicije*. Z "deklaracijo" moramo v nekaterih jezikih povedati, da bomo imeli spremenljivko s takšnim in takšnim imenom in tipom. Z "definicijo" teh spremenljivk damo vrednosti. Python je drugačen, saj nima spremenljivk temveč imena; ime nima tipa in pravzaprav "nima vrednosti", pač pa je ime povezano z vrednostjo.

Kakorkoli. Za lastne potrebe in za potrebe okolij lahko "deklariramo", da bo neko ime povezano z vrednostmi določenega tipa (ali, kot bi si predstavljali prišleki iz drugačnih jezikov, da bo imela spremenljivka vrednosti določenega tipa). Python se ne to ne ozira, te deklaracije lahko, kar se tiče njega, tudi kršimo.

Deklaracija tipa je preprosta. Opravimo jo lahko vnaprej ali sproti, ob določanju vrednosti. Sintaksa je takšna.

```
x: int
```

S tem povemo, da se bo ime `x` nanašalo na reči tipa `int`.

```
x: int = 42
```

S tem povemo, da se bo ime `x` nanašalo na reči tipa `int`, hkrati pa mu že priredimo vrednost.

Malo drugače je - kot bomo videli spodaj - le pri rezultatih funkcij.

Kratek izlet v zgodovino

Za deklaracije tipov obstajata dva sloga. Nemara je kdo vaje C-jevskega

```
int x;
```

C se je tega navzel iz Fortrana (1957), ki je uvedel idejo določanja tipov. Za ta slog pravijo, da je osredotočen na računalnik, ki mu je najbolj pomemben tip ("*imel bom spremenljivko vrste `int` ...*"), ne pa toliko ime ("*... in ime ji bo `x`*").

Pythonov slog,

```
x: int
```

izhaja iz Pascala (1970), ki je poudarjal berljivost. Za človeka je pomembnejše ime ("*imel bom spremenljivko z imenom `x`*") in šele potem tip ("*ki bo vrste `int`*").

Prva oblika je še vedno običajna v jezikih, ki so tudi po siceršnji sintaksi podobni C-ju ali pa celo izhajajo iz njega (Java, C#), v zadnjih desetletjih pa bolj prevladuje druga. V svojih programerskih karierah jo boste gotovo srečali vsaj v TypeScriptu, kar verjetno tudi v Rustu, ki ima popolnoma enako sintakso, pa v Kotlinu, Go-ju...

Kaj in kje

Tipe lahko določamo spremenljivkam, argumentom in rezultatom funkcij, ujetim izjemam...

```
def razmerje(a: int, b: int) -> float:
    return a / b
```

Argumentom funkcij smo določili tipe tako kot spremenljivkam. Rezultat pa smo opisali z `-> float`. Sintaksa : `float` tu ne bi delovala, saj ima dvopičje v glavi funkcije že drugo vlogo.

Sestavljeni tipi

Seznam `int`-ov opišemo z

```
razdalje: list[int]
```

Ne spreglejte, da uporabljamo oglate oklepaje. Okrogli bi pomenili klic.

Slovar, katerega ključi so nizi, vrednosti pa `int`-i, je

```
imenik: dict[str, int]
```

Za terke v osnovi predpostavimo, da so fiksne dolžine in naštejemo tipe njenih elementov. Terka, ki vsebuje niz in dva `int`-a, je

```
kraj: tuple[str, int, int]
```

Terka, ki vsebuje poljubno število elementov določenega tipa, na primer `float`, pa je

```
cene: tuple[float, ...]
```

Te stvari je seveda mogoče gnezditi:

```
zbirka: dict[str, tuple[str, int, list[float]]]
```

Unije tipov

Zgodi se (dobro pa je, da se zgodi čim manjkrat), da je za določeno ime možno, da se bo nanašala na različne tipe. Takšna imena označimo s tipom, ki ga uvozimo iz modula `typing: Union`. Naštejemo mu možne tipe:

```
from typing import Union
```

```
tri: Union[int, float, bool]
```

Slovar, katerega ključi so nizi, vrednosti pa `int` ali `bool`, je

```
imenik: dict[str, Union[int, bool]]
```

Pogosteje se bo zgodilo, da bo vrednost znana ali ne; če bo neznana, bo `None`. Vrednost `None` smemo uporabiti tudi kot tip.

```
imenik: dict[str, Union[int, None]]
```

Tega sicer ne počnemo, saj imao bližnjico: kadar se neko ime nanaša bodisi na reč določenega tipa bodisi na `None`, uporabimo tip `Optional`, ki ga prav tako uvozimo z `typing`.

```
from typing import Optional
```

```
imenik: dict[str, Optional[int]]
```

Abstraktni tipi

Kako določiti tip argumentu te funkcije?

```
def zlepi(s):  
    v = ""  
    for x in s:  
        v += x  
    return v
```

Lahko bi rekli, da je `s: list[str]`, vendar bi funkciji s tem naredili krivico: `s` je lahko tudi terka, ali množica (ali celo slovar).

Pisati `s: Union[list[int], tuple[int, ...], set[int]]` bi bilo nepraktično, pisati `s: Union[list, set, tuple][int]` pa narobe, ker se `Union` ne da uporabljati na ta način.

Ne samo to: funkcija kot argument prebavi tudi datoteko:

```
zlepi(open("stevilke.txt"))  
  
'3\n5\n8\n4\n2\n8\n3\n10\n11\n5\n13\n4\n'
```

Funkcija sprejme karkoli, čez kar je možno iti z zanko `for`. Pravilna anotacija funkcije je

```
from collections.abc import Iterable
```

```
def zlepi(s: Iterable[str]):  
    v = ""  
    for x in s:  
        v += x  
    return v
```

`Iterable` je abstraktni osnovni tip (*abstract base class*), ki ga dobimo v modulu `collections.abc`. Pomeni točno to, kar potrebujemo "nekaj, čez kar je možno iti z zanko". (To pravzaprav ni čisto točno. Pomeni "nekaj, kar zna vrniti iterator".) Ker smo poleg tega dodali `[str]`, smo zahtevali še, naj zanka prek te reči vrača nize.

Takšnih tipov je še *zelo* veliko. Dokumentacija jih opisuje precej tehnično - z metodami, ki jih podpirajo. Tako je najosnovnejši tip, `Container` opisan s tem,

da vsebuje metodo `__contains__`. Če imamo

```
from collections.abc import Container
```

```
s: Container[int]
```

to pomeni, da bomo smeli v zvezi s `s`-jem uporabljati pogoj `x in s`, pri čemer bo moral biti `x` vrste `int`. `Hashable` so stvari, ki jih lahko uporabimo kot ključ v slovarju ali damo v množico, `Sized` so stvari, za katere lahko pokličemo `len...`

Nekatere od teh stvari so res zgolj za type annotation nazije.

Generiki

In če smo že pri njih, pokažimo še generike.

```
def vsota[T](s: Iterable[T]) -> T:
    v: Optional[T] = None
    for x in s:
        if v is None:
            v = x
        else:
            v += x
    return v
```

`vsota` je funkcija, ki prejme `s`, čez katerega je možno iterirati z zanko `for`. `s` bo pri iteriranju vračal elemente tipa `T`; `T` bo tudi rezultat funkcije. `v` bo bodisi `T` bodisi `None`. Ah, da, odkod pa se je vzel `T`? No, `T` je pa pač nek tip - mogoče `float`, mogoče `int`, mogoče `str`. Uvedli smo ga s `[T]` v `vsota[T]`.

Tole sicer še ni čisto popolno: povedati bi morali še, da je `T` neka stvar, ki se jo da seštevati. Če je `T` slučajno slovar (in naše oznake tega ne prepovedujejo!), bo funkcija javila napako v `v += x`.

To se menda da, vendar tega nisem še nikoli počel. In tudi ne mislim. :)