20. Funkcie a parametre

video prezentácia

funkcie a parametre

Parametre funkcií

Chceli by sme zapísať funkciu, ktorá by počítala súčin ľubovoľného počtu nejakých čísel. Aby sme ju mohli volať s rôznym počtom parametrov, využijeme náhradné hodnoty:

```
def sucin(a=1, b=1, c=1, d=1, e=1):
    return a * b * c * d * e
```

Túto funkciu môžeme volať aj bez parametrov, ale nefunguje viac ako 5 parametrov:

```
>>> sucin(3, 7)
    21
>>> sucin(2, 3, 4)
    24
>>> sucin(2, 3, 4, 5, 6)
    720
>>> sucin(2, 3, 4, 5, 6, 7)
    ...
    TypeError: sucin() takes from 0 to 5 positional arguments but 6 were given
>>> sucin()
    1
>>> sucin(13)
    13
```

Ak chceme použiť aj väčší počet parametrov, môžeme využiť zoznam, resp. ľubovoľnú postupnosť:

```
def sucin(postupnost):
    vysl = 1
    for prvok in postupnost:
        vysl *= prvok
    return vysl
```

Teraz to funguje pre ľubovoľný počet čísel, ale musíme ich uzavrieť do hranatých alebo okrúhlych zátvoriek:

```
>>> sucin([3, 7])
    21
>>> sucin([2, 3, 4, 5, 6])
    720
>>> sucin((2, 3, 4, 5, 6, 7))
    5040
```

Namiesto zoznamu môžeme ako parameter poslať aj `range(2, 8)`, t.j. ľubovoľnú štruktúru, ktorá sa dá rozobrať pomocou for-cyklu:

```
>>> sucin(range(2, 8))
    5040
>>> sucin(range(2, 41))
    815915283247897734345611269596115894272000000000
```

Zbalené a rozbalené parametre

Predchádzajúce riešenie stále nerieši náš problém: funkciu s ľubovoľným počtom parametrov. Na toto slúžia tzv. **zbalené parametre** (po anglicky *packing arguments*):

- pred menom parametra v hlavičke funkcie píšeme znak `*` (zvyčajne je to posledný parameter)
- pri volaní funkcie sa všetky zvyšné parametre **zbalia** do jednej n-tice (typ `tuple`)

Otestujme:

```
def test(prvy, *zvysne):
    print('prvy =', prvy)
    print('zvysne =', zvysne)
```

po spustení:

```
>>> test('jeden', 'dva', 'tri')
    prvy = jeden
    zvysne = ('dva', 'tri')
>>> test('jeden')
    prvy = jeden
    zvysne = ()
```

Funkcia sa môže volať s jedným alebo aj viac parametrami. Prepíšme funkciu `sucin()` s použitím jedného zbaleného parametra:

```
def sucin(*ntica):            # zbalený parameter
    vysl = 1
    for prvok in ntica:
        vysl *= prvok
    return vysl
```

Uvedomte si, že teraz jeden parameter `ntica` zastupuje **ľubovoľný počet parametrov** a Python nám do tohto parametra automaticky zbalí všetky skutočné parametre ako jednu n-ticu (`tuple`). Otestujeme:

```
>>> sucin()
    1
>>> sucin(3, 7)
    21
>>> sucin(2, 3, 4, 5, 6, 7)
    5040
>>> sucin(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
    479001600
>>> sucin(range(2, 13))
    ...
    TypeError: unsupported operand type(s) for *=: 'int' and 'range'
```

V poslednom príklade vidíte, že `range(...)` tu nefunguje: Python tento jeden parameter zbalí do jednoprvkovej n-tice a potom sa s týmto `range()` bude chcieť násobiť, čo samozrejme nefunguje.

Teraz sa pozrime na ďalší príklad, ktorý ilustruje možnosti práce s parametrami funkcií. Napíšme funkciu, ktorá dostáva dva alebo tri parametre a nejako ich vypíše:

```
def pis(meno, priezvisko, rok=2018):
    print(f'volam sa {meno} {priezvisko} a narodil som sa v {rok}')
```

Napríklad:

```
>>> pis('Janko', 'Hrasko', 2014)
    volam sa Janko Hrasko a narodil som sa v 2014
>>> pis('Juraj', 'Janosik')
    volam sa Juraj Janosik a narodil som sa v 2018
```

Malá nepríjemnosť nastáva vtedy, keď máme takéto hodnoty pripravené v nejakej štruktúre:

```
>>> p1 = ['Janko', 'Hrasko', 2014]
>>> p2 = ['Juraj', 'Janosik']
>>> p3 = ['Monty', 'Python', 1968]
>>> pis(p1)
    ...
    TypeError: pis() missing 1 required positional argument: 'priezvisko'
```

Túto funkciu nemôžeme volať s trojprvkovým zoznamom, ale musíme prvky tohto zoznamu **rozbaliť**, aby sa priradili do príslušných parametrov, napríklad:

```
>>> pis(p1[0], p1[1], p1[2])
    volam sa Janko Hrasko a narodil som sa v 2014
>>> pis(p2[0], p2[1])
    volam sa Juraj Janosik a narodil som sa v 2018
```

Takáto situácia sa pri programovaní stáva dosť často: v nejakej štruktúre (napríklad v zozname) máme pripravené parametre pre danú funkciu a my potrebujeme túto funkciu zavolať s rozbalenými prvkami štruktúry. Na toto slúži **rozbaľovací operátor**, pomocou ktorého môžeme ľubovoľnú štruktúru poslať ako skupinu parametrov, pričom sa automaticky rozbalia (a teda prvky sa priradia do formálnych parametrov). Rozbaľovací operátor pre parametre je opäť znak `*` a používa sa takto:

```
>>> pis(*p1)              # je to isté ako pis(p1[0], p1[1], p1[2])
    volam sa Janko Hrasko a narodil som sa v 2014
>>> pis(*p2)              # je to isté ako pis(p2[0], p2[1])
    volam sa Juraj Janosik a narodil som sa v 2018
```

Takže, všade tam, kde sa očakáva nie jedna štruktúra ako parameter, ale veľa parametrov, ktoré sú prvkami tejto štruktúry, môžeme použiť tento rozbaľovací operátor (po anglicky *unpacking argument lists*).

Tento operátor môžeme využiť napríklad aj v takýchto situáciách:

```
>>> print(range(10))
    range(0, 10)
>>> print(*range(10))
    0 1 2 3 4 5 6 7 8 9
>>> print(*range(10), sep='...')
    0...1...2...3...4...5...6...7...8...9
>>> param = (3, 20, 4)
>>> print(*range(*param))
    3 7 11 15 19
>>> dvenasto = 2 ** 100
>>> print(dvenasto)
    1267650600228229401496703205376
>>> print(*str(dvenasto))
    1 2 6 7 6 5 0 6 0 0 2 2 8 2 2 9 4 0 1 4 9 6 7 0 3 2 0 5 3 7 6
>>> print(*str(dvenasto), sep='-')
    1-2-6-7-6-5-0-6-0-0-2-2-8-2-2-9-4-0-1-4-9-6-7-0-3-2-0-5-3-7-6
>>> p = [17, 18, 19, 20, 21]
```

```
>>> [*p[3:], *range(5), *p]
    [20, 21, 0, 1, 2, 3, 4, 17, 18, 19, 20, 21]
```

Pripomeňme si funkciu `sucin()`, ktorá počítala súčin ľubovoľného počtu čísel - tieto sa spracovali jedným zbaleným parametrom. Teda funkcia očakáva veľa parametrov a niečo z nich vypočíta. Ak ale máme jednu štruktúru, ktorá obsahuje tieto čísla, môžeme použiť rozbaľovací operátor:

```
>>> cisla = [7, 11, 13]
>>> sucin(cisla)              # zoznam [7, 11, 13] sa násobí 1
    [7, 11, 13]
>>> sucin(*cisla)             # sucin(cisla[0], cisla[1], cisla[2])
    1001
>>> sucin(*range(2, 11))
    3628800
```

Parameter s meniteľnou hodnotou

Teraz trochu odbočíme od zbalených a rozbalených parametrov. Ukážme veľký problém, ktorý nás môže zaskočiť v situácii, keď náhradnou hodnotou parametra je meniteľný typ (mutable). Pozrime na túto nevinne vyzerajúcu funkciu:

```
def pokus(a=1, b=[]):
    b.append(a)
    return b
```

Očakávame, že ak neuvedieme druhý parameter, výsledkom funkcie bude jednoprvkový zoznam s prvkom prvého parametra. Skôr, ako to otestujeme, vypíšme, ako túto našu funkciu vidí `help()`:

```
>>> help(pokus)
    Help on function pokus in module __main__:

    pokus(a=1, b=[])
```

a teraz test:

```
>>> pokus(2)
    [2]
```

Zatiaľ je všetko v poriadku. Ale po druhom spustení:

```
>>> pokus(7)
    [2, 7]
```

Vidíme, že Python si tu nejako pamätá aj naše prvé spustenie tejto funkcie. Znovu pozrime `help()`:

```
>>> help(pokus)
    Help on function pokus in module __main__:

    pokus(a=1, b=[2, 7])
```

A vidíme, že sa dokonca zmenila hlavička našej funkcie `pokus()`. Mali by sme teda rozumieť, čo sa tu vlastne deje:

- Python si pre každú funkciu pamätá zoznam všetkých **náhradných hodnôt** pre formálne parametre funkcie, tak ako sme ich zadefinovali v hlavičke (môžete si pozrieť premennú `pokus.__defaults__`)
- ak sú v tomto zozname len nemeniteľné hodnoty (immutable), nevzniká žiaden problém

- problémom sú meniteľné hodnoty (mutable) v tomto zozname: pri volaní funkcie, keď treba použiť náhradnú hodnotu, Python použije hodnotu z tohto zoznamu (použije referenciu na túto štruktúru) - keď tomuto parametru ale v tele funkcie zmeníme obsah, zmení sa tým aj hodnota v zozname náhradných hodnôt (`pokus.__defaults__`)

Z tohto pre nás vyplýva, že radšej nikdy nebudeme definovať náhradnú hodnotu parametra ako meniteľný objekt. Funkciu `pokus` by sme mali radšej zapísať takto:

```
def pokus(a=1, b=None):
    if b is None:
        b = []
    b.append(a)
    return b
```

A všetko by fungovalo tak, ako sme očakávali.

Skúsení programátori vedia túto vlastnosť využiť veľmi zaujímavo. Napríklad do funkcie posielame nejaké hodnoty a funkcia nám oznamuje, či už sa taká vyskytla, alebo ešte nie:

```
def kontrola(hodnota, bola=set()):
    if hodnota in bola:
        print(hodnota, 'uz bola')
    else:
        bola.add(hodnota)
        print(hodnota, 'OK')
```

a test:

```
>>> kontrola(7)
    7 OK
>>> kontrola(17)
    17 OK
>>> kontrola(-7)
    -7 OK
>>> kontrola(17)
    17 uz bola
>>> kontrola(7)
    7 uz bola
```

Tento test funguje bez globálnej premennej (my už teraz vieme, že to funguje vďaka „tajnej“ premennej vo funkcii).

Zbalené pomenované parametre

Pozrime sa na túto funkciu:

```
def vypis(meno, vek, vyska, vaha, bydlisko):
    print('volam sa', meno)
    print('    vek =', vek)
    print('    vyska =', vyska)
    print('    vaha =', vaha)
    print('    bydlisko =', bydlisko)
```

otestujeme:

```
>>> vypis('Janko Hrasko', vek=5, vyska=7, vaha=0.3, bydlisko='Pri poli')
    volam sa Janko Hrasko
        vek = 5
        vyska = 7
        vaha = 0.3
```

```
        bydlisko = Pri poli
```

Radi by sme aj tu dosiahli podobnú vlastnosť parametrov, ako to bolo pri zbalenom parametri, ktorý do jedného parametra dostal ľubovoľný počet skutočných parametrov. V tomto prípade by sme ale chceli, aby sa takto zbalili všetky vlastnosti vypisovanej osoby ale aj s príslušnými menami týchto vlastností. V tomto prípade nám pomôžu **zbalené pomenované parametre** (po anglicky *keyword argument packing*): namiesto viacerých pozičných parametrov, uvedieme jeden s dvomi hviezdičkami `**`:

```
def vypis(meno, **vlastnosti):
    print('volam sa', meno)
    for k, h in vlastnosti.items():
        print('    ', k, '=', h)
```

Tento zápis označuje, že ľubovoľný počet pomenovaných parametrov sa zbalí do jedného parametra a ten vo vnútri funkcie bude typu **slovník** (asociatívne pole `dict`). Uvedomte si ale, že v slovníku sa nezachováva poradie dvojíc:

```
>>> vypis('Janko Hrasko', vek=5, vyska=7, vaha=0.3, bydlisko='Pri poli')
    volam sa Janko Hrasko
         vyska = 7
         vaha = 0.3
         bydlisko = Pri poli
         vek = 5
```

Ďalší príklad tiež ilustruje takýto zbalený slovník:

```
import tkinter

def kruh(r, x, y):
    canvas.create_oval(x-r, y-r, x+r, y+r)

canvas = tkinter.Canvas()
canvas.pack()

kruh(50, 100, 100)
```

Funkcia `kruh()` definuje nakreslenie kruhu s daným polomerom a stredom, ale nijako nevyužíva ďalšie parametre na definovanie farieb a obrysu kruhu. Doplňme do funkcie zbalené pomenované parametre:

```
def kruh(r, x, y, **param):
    canvas.create_oval(x-r, y-r, x+r, y+r)
```

Toto označuje, že `kruh()` môžeme zavolať s ľubovoľnými ďalšími pomenovanými parametrami, napríklad `kruh(..., fill='red', width=7)`. Tieto parametre ale chceme ďalej poslať do funkcie `create_oval()`. Určite sem nemôžeme poslať `param`, lebo toto je premenná typu `dict` a `create_oval()` s tým pracovať nevie. Tu by sa nám zišlo premennú `param` rozbaliť do viacerých pomenovaných parametrov: Rozbaľovací operátor pre pomenované parametre sú dve hviezdičky `**`, teda zapíšeme:

```
def kruh(r, x, y, **param):
    canvas.create_oval(x-r, y-r, x+r, y+r, **param)
```

a teraz funguje aj

```
kruh(50, 100, 100)
kruh(30, 150, 100, fill='red')
kruh(100, 200, 200, width=10, outline='green')
```

Takýto rozbaľovací parameter by sme vedeli využiť aj v predchádzajúcom príklade s funkciou `vypis()`:

```
>>> p1 = {'meno':'Janko Hrasko', 'vek':5, 'vyska':7, 'vaha':0.3, 'bydlisko':'Pri poli'}
>>> vypis(**p1)
    volam sa Janko Hrasko
         vaha = 0.3
         vek = 5
         vyska = 7
         bydlisko = Pri poli
>>> p2 = {'vek':25, 'narodeny':'Terchova', 'popraveny':'Liptovsky Mikulas'}
>>> vypis('Juraj Janosik', **p2)
    volam sa Juraj Janosik
         popraveny = Liptovsky Mikulas
         vek = 25
         narodeny = Terchova
```

Funkcia ako hodnota

v Pythone sú aj funkcie objektmi a môžeme ich priradiť do premennej, napríklad:

```
>>> def fun1(x): return x * x
>>> fun1(7)
    49
>>> cojaviem = fun1
>>> cojaviem(8)
    64
```

Funkcie môžu byť prvkami zoznamu, napríklad:

```
>>> def fun2(x): return 2*x + 1
>>> def fun3(x): return x // 2
>>> zoznam = [fun1, fun2, fun3]
>>> for f in zoznam:
        print(f(10))
    100
    21
    5
```

Funkciu môžeme poslať ako parameter do inej funkcie, napríklad:

```
>>> def urob(fun, x):
        return fun(x)
>>> urob(fun2, 3.14)
    7.28
```

Funkcia (teda referencia na funkciu) môže byť aj prvkom slovníka. Pekne to ilustruje príklad s korytnačkou:

```
def vykonaj():
    t = turtle.Turtle()
    p = {'fd': t.fd, 'rt': t.rt, 'lt': t.lt}
    while True:
        prikaz, parameter = input('> ').split()
        p[prikaz](int(parameter))
```

a funguje napríklad:

```
>>> vykonaj()
```

```
    > fd 100
    > lt 90
    > fd 50
    > rt 60
    > fd 100
```

Anonymné funkcie

Často sa namiesto jednoriadkovej funkcie, ktorá počíta jednoduchý výraz a tento vráti ako výsledok (`return`) používa špeciálna konštrukcia `lambda`. Tá vygeneruje tzv. anonymnú funkciu, ktorú môžeme priradiť do premennej alebo poslať ako parameter do funkcie, napríklad:

```
>>> urob(lambda x: 2*x + 1, 3.14)
    7.28
```

Tvar konštrukcie `lambda` je nasledovný:

```
lambda parametre: výraz
```

Tento zápis nahrádza definovanie funkcie:

```
def anonymne_meno(parametre):
    return vyraz
```

Môžeme zapísať napríklad:

```
lambda x: x % 2==0              # funkcia vráti True pre párne číslo
lambda x, y: x ** y             # vypočíta príslušnú mocninu čísla
lambda x: isinstance(x, int)    # vráti True pre celé číslo
```

Mapovacie funkcie

Ideu funkcie ako parametra najlepšie ilustruje takáto funkcia `mapuj()`:

```
def mapuj(fun, postupnost):
    vysl = []
    for prvok in postupnost:
        vysl.append(fun(prvok))
    return vysl
```

Funkcia aplikuje danú funkciu (prvý parameter) na všetky prvky nejakej postupnosti (zoznam, n-tica, …) a z výsledkov poskladá nový zoznam, napríklad:

```
>>> mapuj(fun1, (2, 3, 7))
    [4, 9, 49]
>>> mapuj(list, 'Python'))
    [['P'], ['y'], ['t'], ['h'], ['o'], ['n']]
>>> mapuj(lambda x: [x] * x, range(1, 6))
    [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4], [5, 5, 5, 5, 5]]
```

V Pythone existuje štandardná funkcia `map()`, ktorá robí skoro to isté ako naša funkcia `mapuj()` ale s tým rozdielom, že `map()` nevracia zoznam, ale niečo ako generátorový objekt, ktorý môžeme použiť ako prechádzanú postupnosť vo for-cykle, alebo napríklad pomocou `list()` ho previesť na zoznam, napríklad:

```
>>> list(map(int, str(2 ** 30)))
    [1, 0, 7, 3, 7, 4, 1, 8, 2, 4]
```

Vráti zoznam cifier čísla `2**30`.

Podobná funkcii `mapuj()` je aj funkcia `filtruj()`, ktorá z danej postupnosti (iterovateľný objekt) vyrobí nový zoznam, ale nechá v ňom len tie prvky, ktoré spĺňanú nejakú podmienku. Podmienka je definovaná funkciou, ktorá je prvým parametrom:

```
def filtruj(fun, postupnost):
    vysl = []
    for prvok in postupnost:
        if fun(prvok):
            vysl.append(prvok)
    return vysl
```

Napríklad:

```
>>> def podm(x):               # zistí, či je číslo párne
        return x % 2==0
>>> list(range(1, 20, 3))
    [1, 4, 7, 10, 13, 16, 19]
>>> mapuj(podm, range(1, 20, 3))
    [False, True, False, True, False, True, False]
>>> filtruj(podm, range(1, 20, 3))
    [4, 10, 16]
```

Podobne ako pre `mapuj()` existuje štandardná funkcia `map()`, aj pre `filtruj()` existuje štandardná funkcia `filter()` - tieto dve funkcie ale nevracajú zoznam (`list`) ale postupnosť, ktorá sa dá prechádzať for-cyklom alebo poslať ako parameter do funkcie, kde sa očakáva postupnosť.

Ukážkovým využitím funkcie `map()` je funkcia, ktorá počíta ciferný súčet nejakého čísla:

```
def cs(cislo):
    return sum(map(int, str(cislo)))

>>> cs(1234)
    10
```

Generátorová notácia

Veľmi podobná funkcii `map()` je generátorová notácia (po anglicky **list comprehension**):

- je to spôsob, ako môžeme elegantne vygenerovať nejaký zoznam pomocou for-cyklu a nejakého výrazu
- do hranatých zátvoriek `[...]` nezapíšeme prvky zoznamu, ale predpis, akým sa majú tieto prvky vytvoriť
- základný tvar tohto zápisu je:

```
- [vyraz for i in postupnost]
```

- kde `výraz` najčastejšie obsahuje premennú cyklu a `postupnosť` je ľubovoľná štruktúra, ktorá sa dá prechádzať for-cyklom (napríklad `list`, `set`, `str`, `range()`, riadky otvoreného súboru, ale aj výsledok `map()` a `filter()` a pod.)
- táto notácia môže používať aj vnorené cykly ale aj podmienku `if`, vtedy je to v takomto tvare:

```
[vyraz for i in postupnost if podmienka]
```

alebo

```
[vyraz for i in postupnost for j in postupnost]
```

alebo

```
[vyraz for i in postupnost for j in postupnost if podmienka]
```

a podobne

- generátorová notácia s podmienkou nechá vo výsledku len tie prvky, ktoré spĺňajú danú podmienku

Niekoľko príkladov:

```
>>> [i ** 2 for i in range(1, 11)]
    [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

je to isté ako:

```
>>> vysl = []
>>> for i in range (1, 11):
        vysl.append(i ** 2)
>>> vysl
    [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Vnorené cykly:

```
>>> [i * j for i in range(1, 5) for j in range(1, 5)]
    [1, 2, 3, 4, 2, 4, 6, 8, 3, 6, 9, 12, 4, 8, 12, 16]
```

je to isté ako:

```
>>> vysl = []
>>> for i in range(1, 5):
        for j in range(1, 5):
            vysl.append(i * j)
>>> vysl
    [1, 2, 3, 4, 2, 4, 6, 8, 3, 6, 9, 12, 4, 8, 12, 16]
```

Ďalšie cykly:

```
>>> [[i * j for i in range(1, 5)] for j in range(1, 5)]
    [[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
>>> [[i * j] for i in range(1, 5) for j in range(1, 5)]
    [[1], [2], [3], [4], [2], [4], [6], [8], [3], [6], [9], [12], [4], [8], [12], [16]]
```

Generátorová notácia s podmienkou:

```
>>> [i for i in range(100) if cs(i) == 5]        # cs() vypočíta ciferný súčet
    [5, 14, 23, 32, 41, 50]
```

Tento zápis nájde všetky čísla z intervalu `<0, 99>`, ktorých ciferný súčet je 5. Doteraz by sme to zapisovali takto:

```
>>> vysl = []
>>> for i in range(100):
        if cs(i) == 5:
            vysl.append(i)
>>> vysl
    [5, 14, 23, 32, 41, 50]
```

Generátorová notácia ako parameter do `join` (nepoužili sme tu hranaté zátvorky):

```
>>> ''.join(2 * znak for znak in 'python')
    'ppyytthhoonn'
```

Pomocou tejto konštrukcie by sme vedeli zapísať aj mapovacie funkcie:

```
def mapuj(fun, zoznam):
    return [fun(prvok) for prvok in zoznam]

def filtruj(fun, zoznam):
    return [prvok for prvok in zoznam if fun(prvok)]
```

Všimnite si, že aj funkcia `filtruj()` využíva `if`, ktorý je vo vnútri generátorovej notácie.

Cvičenia

L.I.S.T.

- riešenia **aspoň 15 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- pozri si Riešenie úloh 20. cvičenia

1. Napíš funkciu `max()`, ktorá môže mať ľubovoľný počet parametrov. Funkcia zistí maximálnu hodnotu. Ak je prázdny počet parametrov, funkcia vyvolá chybu `'TypeError: max expected 1 arguments, got 0'`. Nepoužívaj štandardnú funkciu `max`. Napríklad:

```
>>> max(9, 13, 11)
    13
>>> max(*'python')
    'y'
>>> max()
    ...
    TypeError: max expected 1 arguments, got 0
```

   Do funkcie `max()` pridaj takéto správanie: v prípade, že má zadaný iba jeden parameter a tento je neprázdny `list`, `tuple` alebo `set`, vráti maximálnu hodnotu z tejto štruktúry. Inak bude pracovať tak, ako doteraz. Napríklad:

```
>>> max((3, 'a'), (3, 'b'), (2, 'x'))
    (3, 'b')
>>> max(9, 13, 11)
    13
>>> p = (9, 13, 11)
>>> max(p)
```

```
    13
>>> zoz = list('python')
>>> max(zoz)
    'y'
>>> max([])
    ...
    TypeError: max expected 1 arguments, got 0
```

Nepoužívaj štandardnú funkciu `max` (môžeš otestovať, ako sa v týchto prípadoch správa štandardná funkcia `max`).

2. Napíš funkciu `spoj(...)` s ľubovoľným počtom parametrov, pričom, ak sú všetky typu `list`, výsledkom bude zreťazenie všetkých týchto parametrov. Ak aspoň jeden z parametrov nie je `list`, výsledkom funkcie bude `None`. Napríklad:

```
>>> z = spoj(['a', 1], [], [('b', 2)])
>>> z
    ['a', 1, ('b', 2)]
>>> spoj()
    []
>>> print(spoj(['a', 1], [], ('b', 2)))
    None
```

3. Napíš funkciu `vypis(postupnost)`, ktorá pomocou `print` vypíše všetky prvky danej postupnosti (zoznam, n-tica, …) do jedného riadka, prvky sú oddelené znakmi `', '`. Nepouži žiadne cykly (asi využiješ len parametre `print`). Napríklad:

```
>>> vypis([123, 'ahoj', (50, 120), 3.14])
    123, ahoj, (50, 120), 3.14
>>> vypis(range(3, 10, 2))
    3, 5, 7, 9
>>> vypis('Python')
    P, y, t, h, o, n
```

4. Funkciu z predchádzajúcej úlohy prerob na funkciu `retazec(postupnost)`, ktorá namiesto výpisu, zostaví znakový reťazec a tento vráti ako výsledok. Ak si to vyriešil/a pomocou cyklu, skús to vyriešiť aj bez cyklov len pomocou `map` a `join`. Napríklad:

```
>>> r = retazec([123, 'ahoj', (50, 120), 3.14])
>>> r
    '123, ahoj, (50, 120), 3.14'
>>> retazec(range(3, 10, 2))
    '3, 5, 7, 9'
```

5. Napíš funkciu `aplikuj(...)`, ktorej parametrami sú nejaké funkcie, okrem posledného parametra, ktorým je nejaká hodnota. Funkcia postupne zavolá všetky tieto funkcie s danou hodnotou, pričom každú ďalšiu funkciu aplikuje na predchádzajúci výsledok.
   Napríklad `aplikuj(f1, f2, f3, x)` vypočíta `f3(f2(f1(x)))`. Funkcia by mala správne pracovať pre ľubovoľný nenulový počet parametrov. Napríklad:

```
>>> aplikuj(float, int, str, '-314159e-3')
    '-314'
>>> def rev(x): return x[::-1]
>>> aplikuj(str, rev, int, 1074)
    4701
```

```
>>> aplikuj(abs, lambda x: x+7, -17)
    24
```

6. Do funkcie `max()` z úlohy (1) pridaj na koniec pomenovaný parameter `key` s náhradnou hodnotou `None`. Teraz bude funkcia pracovať takto:
   - v prípade, že `key` má hodnotu `None`, bude pracovať rovnako ako v úlohe (1)
   - inak predpokladáme, že `key` je daná funkcia s jedným parametrom, vďaka tejto funkcii bude `max` hľadať taký prvok `x`, pre ktorý je `key(x)` maximálny

   Zapíš:

```
def max(*post, key=None):
    ...
```

   Napríklad:

```
>>> max(3, 7, 11, 4)
    11
>>> max(3, 7, 11, 4, key=lambda x: -x)
    3
>>> max([3, 7, 11, 4], key=str)
    7
```

   Nepoužívaj štandardnú funkciu `max` (aj štandardná funkcia `max` funguje presne takto s pomenovaným parametrom `key`, môžeš to otestovať).

7. Napíš funkciu `najdlhsi(*retazec)`, ktorá pre ľubovoľný počet reťazcov vráti najdlhší z nich. Napríklad:

```
>>> najdlhsi('a', '', 'bc', 'd', 'ef')
    'bc'
>>> najdlhsi(*'mam rad programovanie v pythone'.split())
    'programovanie'
```

   Vyrieš najprv pomocou for-cyklu vo funkcii:

```
def najdlhsi(*retazec):
    ...
    for ...
        ...
    return ...
```

   Vyrieš pomocou štandardnej funkcie `max()` a parametra `key` (prípadne `max` z predchádzajúcej úlohy):

```
def najdlhsi(*retazec):
    return max(...)
```

   Otestuj funkciu na textovom súbore: otvor nejaký textový súbor a zavolaj túto funkciu tak, aby ti vrátila najdlhší riadok (alebo slovo) z tohto súboru.

8. Napíš funkciu `map2(fun, param1, param2)`, ktorá bude pracovať podobne ako funkcia `mapuj()` z prednášky, len funkcia `fun` očakáva dva parametre: jeden z postupnosti `param1` a druhý z postupnosti `param2`. Ak majú tieto postupnosti rôznu dĺžku, tak berie len počet kratšej z nich. Nepouži štandardnú funkciu `map`. Napríklad:

```
>>> def f(x, y): return x * y
>>> map2(f, 'python', range(1, 6))
    ['p', 'yy', 'ttt', 'hhhh', 'ooooo']
>>> map2(f, ('a', 4, (1, 2)), [3, 5, 2])
    ['aaa', 20, (1, 2, 1, 2)]
```

Otestuj, či takto funguje aj štandardná funkcia `map` (keď má tri parametre, tak prvým je binárna funkcia, ktorá sa aplikuje na prvky dvoch postupností).

9. Funkcia `kruh()` z prednášky funguje aj bez určovania farby výplne:

```
def kruh(r, x, y, **param):
    canvas.create_oval(x-r, y-r, x+r, y+r, **param)
```

Doplň funkciu tak, aby každý takto kreslený kruh bol vyplnený buď farbou udanou v parametroch alebo bude inak červený, podobne, ak nie je daná hrúbka obrysu (`width`), nastaví sa hrúbka 3. Napríklad:

```
>>> kruh(100, 100, 100, outline='blue', width=1)    # červený s hrúbkou 1
>>> kruh(30, 50, 100, width=3, fill='blue')         # modrý s hrúbkou 3
```

10. Funkcia `vykonaj()` z prednášky spadne pri chybnom mene príkazu alebo chybnom parametri:

```
def vykonaj():
    t = turtle.Turtle()
    p = {'fd': t.fd, 'rt': t.rt, 'lt': t.lt}
    while True:
        prikaz, parameter = input('> ').split()
        p[prikaz](int(parameter))
```

Oprav ju tak, aby nespadla, ale vypísala sa o tom správa a ďalej sa pokračovalo. Využi metódu `get()` pre slovník, ktorá vyrieši situáciu so zle zadaným menom príkazu tak, že sa zavolá anonymná funkcia, ktorá vypíše správu (napríklad `lambda: print('chybne meno prikazu', ...)`). Napríklad:

```
>>> vykonaj()
    > fd 100
    > bk 50
    chybne meno prikazu 'bk'
    > rt 90
    > lt 45x
    chybny parameter '45x'
    >
```

11. Napíš funkciu `mnozina(n)`, ktorá vráti množinu čísel, ktoré sú druhými mocninami mínus `1` čísel od 1 do `n`. Úlohu vyrieš dvoma rôznymi spôsobmi:

```
def mnozina(n):
    return { ... for ...}

def mnozina(n):
    return set(map( ... ))
```

```
>>> mnozina(4)
    {0, 3, 8, 15}
```

12. Napíš funkciu `prevrat_slova(veta)`, ktorá vráti zadanú vetu tak, že každé slovo v nej bude otočené. Napríklad:

```
>>> prevrat_slova('isiel macek do malacek')
    'leisi kecam od kecalam'
```

Zapíš funkciu pomocou jediného riadka s príkazom `return`:

```
def prevrat_slova(veta):
    return ...
```

13. Napíš funkciu `nahodne(n)`, ktorá vygeneruje `n`-prvkový zoznam náhodných čísel z intervalu `<0, 2*n-1>`. Použi generátorovú notáciu:

```
def nahodne(n):
    return [ ... ]
```

Napríklad:

```
>>> nahodne(4)
    [5, 7, 2, 5]
```

14. Napíš funkciu `matica(n, m, hodnota=0)`, ktorá vygeneruje dvojrozmerný zoznam `n` riadkov po `m` stĺpcov. Prvkami matice budú zadané hodnoty. Použi generátorovú notáciu:

```
def matica(n, m, hodnota=0):
    return [ ... ]
```

Napríklad:

```
>>> m = matica(3, 4, 1)
>>> m
    [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
>>> m[0][2] = 7
>>> m
    [[1, 1, 7, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

15. Napíš funkciu `matica_nahodne(n, m, rozsah=2)`, ktorá vygeneruje dvojrozmerný zoznam `n` riadkov po `m` stĺpcov. Prvkami matice budú náhodné hodnoty z rozsahu `<0, rozsah)`. Použi generátorovú notáciu:

```
def matica_nahodne(n, m, rozsah=2):
    return [ ... ]
```

Napríklad:

```
>>> matica_nahodne(3, 4, 3)
    [[1, 0, 2, 1], [0, 0, 2, 1], [0, 2, 2, 1]]
```

16. Napíš funkciu `zadaj(text)`, ktorá si najprv vypýta so zadaným textom nejaký vstup (`input()`) a potom ho prerobí na zoznam celých čísel. V prípade chyby nespadne, ale vráti prázdny zoznam. Napríklad:

```
>>> zoz = zadaj('zadaj cisla: ')
    zadaj cisla: 6 73 -8
>>> zoz
    [6, 73, -8]
>>> zoz = zadaj('zadaj: ')
    zadaj: 6 73 a -8
>>> zoz
    []
```

Úlohu vyrieš takouto schémou funkcie (nepridávaj ďalšie riadky):

```
def zadaj(text):
    try:
        return ...
    except ...:
        return ...
```

17. Už poznáme štandardnú funkciu `enumerate()`, ktorá z danej postupnosti vracia postupnosť dvojíc. Napíš vlastnú funkciu `enumerate(postupnost)`, ktorá vytvorí takýto zoznam dvojíc (`list` s prvkami `tuple`): prvým prvkom bude poradové číslo dvojice a druhým prvkom prvok zo vstupnej postupnosti. Napríklad:

```
def enumerate(postupnost):
    return ...

>>> enumerate('python')
    [(0, 'p'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

18. Napíš funkciu `zip(p1, p2)`, ktorá z dvoch postupností rovnakých dĺžok vytvorí zoznam zodpovedajúcich dvojíc, t.j. zoznam, v ktorom prvým prvkom bude dvojica prvých prvkov postupností, druhým prvkom dvojica druhých prvkov, atď. … Napríklad:

```
>>> zip('python', [2, 3, 5, 7, 11, 13])
    [('p', 2), ('y', 3), ('t', 5), ('h', 7), ('o', 11), ('n', 13)]
```

Pokús sa to zapísať tak, aby to fungovala aj pre postupnosti rôznych dĺžok: vtedy vytvorí len toľko dvojíc, koľko je prvkov v kratšej z týchto postupností. Napríklad:

```
>>> zip('python', [2, 3, 5, 7, 11])
    [('p', 2), ('y', 3), ('t', 5), ('h', 7), ('o', 11)]
```

Pokús sa to zapísať tak, aby funkcia fungovala pre ľubovoľný počet ľubovoľne dlhých postupností. Napríklad:

```
>>> zip('python', [2, 3, 5, 7, 11], 'abcd')
    [('p', 2, 'a'), ('y', 3, 'b'), ('t', 5, 'c'), ('h', 7, 'd')]
```

Každá z týchto verzii funnkcie `zip` sa dá zapísať v jedinom riadku:

```
def zip( ... ):
    return ...
```

V Pythone existuje **štandardná funkcia** `zip()`, ktorá funguje skoro presne ako táto posledná verzia funkcie, len jej výsledkom nie je zoznam, ale opäť iterovateľná postupnosť (dá sa prechádzať for-cyklom, alebo vypísať pomocou `print(*zip(...))`, alebo previesť na zoznam pomocou `list(zip(...))`). Funkciu `zip()` (štandardnú alebo tú vašu) môžeme použiť aj vo for-cykle. Napríklad:

```
>>> ''.join(x*y for x, y in zip('python', [2, 3, 2, 1, 3, 2]))
    'ppyyytthooonn'
```

19. Zapíš funkciu `enumerate` z úlohy (17) pomocou volania `zip`:

```
20. def enumerate(postupnost):
21.     return zip( ... )
```

11. Týždenný projekt

L.I.S.T.

- riešenie odovzdaj na úlohový server https://list.fmph.uniba.sk/

Napíš pythonovský modul, ktorý bude obsahovať jedinú triedu `Pajton` a žiadne iné globálne premenné, funkcie a triedy:

```
class Pajton:
    def __init__(self):
        self.tab = {}

    def prem(self, meno):
        return ...

    def vyraz(self, retazec):
        return ...

    def prirad(self, meno, hodnota):
        ...

    def prikaz(self, retazec):
        if retazec == 'globals()':
            return self.globals()
        if retazec == 'dir()':
            return self.dir()
        ...

    def dir(self):
        return ...

    def globals(self):
        return ...
```

Táto trieda umožní programovať vo veľmi jednoduchom programovacom jazyku. V tomto jazyku môžeme postupne zadávať priraďovacie príkazy alebo zisťovať hodnoty výrazov podobne ako v IDLE jazyka Python. Tento zjednodušený programovací jazyk zvláda len celočíselnú aritmetiku s operáciami `+`, `-`, `*`, `/`, pričom `/` označuje celočíselné delenie (pythonovské `//`), kde delenie nulou má hodnotu `0`. Aritmetické výrazy môžu byť len týchto dvoch typov:

- jednoduchý operand:

    - celé číslo
    - meno premennej (ak ešte nemala priradenú hodnotu, bude to chyba `NameError`)
- **operand operácia operand … operácia operand**, kde operácie sú niektoré zo symbolov `+`, `-`, `*`, `/`
    - medzi operandmi a operáciou musí byť aspoň jedna medzera, inak je to `SyntaxError`
    - operandy sú len jednoduché operandy (čísla a premenné)
    - aritmetickými výrazmi sú teda aj postupnosti operandov, medzi ktorými sú operácie - takéto výrazy sa budú vyhodnocovať zľava doprava a to bez ohľadu na prioritu operácií; teda výraz `2 + 3 * 4` sa vyhodnotí ako `20`

Priraďovací príkaz má tvar:

- **premenná `=` výraz**
    - znak `=` musí byť oddelený od premennej a výrazu aspoň jednou medzerou, inak je to `SyntaxError`

Ako príkaz môžeme tomuto jazyku zadať:

- aritmetický výraz - vtedy vypíše jeho hodnotu
- priraďovací príkaz - vtedy nič nevypisuje len vykoná priradenie
- príkaz `globals()` - vtedy vypíše hodnoty všetkých premenných, do ktorých sa niečo doteraz priradilo
- príkaz `dir()` - vtedy vypíše množinu všetkých mien premenných

Metódy majú fungovať takto:

- inicializácia `__init__()` vytvorí prázdnu tabuľku premenných, bude to slovník (`dict`), v ktorom kľúčom bude meno a hodnotou bude zodpovedajúca hodnota premennej, nemeň meno atribútu `tab`, lebo testovač bude kontrolovať jeho obsah
- metóda `prem(meno)` vráti hodnotu zodpovedajúcu premennej s daným menom, ak takéto meno v slovníku `self.tab` nenájde, vyvolá výnimku `NameError`
- metóda `vyraz(retazec)` vráti hodnotu aritmetického výrazu (hodnotou bude celé číslo), alebo vyvolá jednu z chýb `NameError` alebo `SyntaxError`
- metóda `prirad(meno, hodnota)` skontroluje korektnosť mena premennej (či nezačína číslicou a obsahuje len písmená, číslice resp. podčiarkovníky) a v slovníku `self.tab` si pre dané meno zapamätá novú hodnotu
- metóda `prikaz(retazec)`, ak je reťazec korektný, zavolá metódy `vyraz` alebo `prirad` a vráti buď hodnotu výrazu (celé číslo) alebo vykoná priradenie a vtedy vráti `None`; okrem týchto prípadov, metóda zrejme vyvolá aj metódy `globals` a `dir`
- metóda `dir()` vráti množinu mien všetkých premenných
- metóda `globals()` vráti viacriadkový reťazec s hodnotami všetkých premenných; každú premennú v tvare `premenná: hodnota`; ak je slovník premenných prázdny, funkcia vráti `None`

Na testovanie môžeš využiť tento kód (umiestni ho za definíciu triedy, môže ostať v module, aj keď ho budeš odovzdávať na testovanie):

```
if __name__ == '__main__':
    p = Pajton()
    while True:
        try:
            hodn = p.prikaz(input('>>> '))
            if hodn is not None:
                print(hodn)
        except SyntaxError:
            print('+++ syntakticka chyba +++')
        except NameError:
            print('+++ chybne meno premennej +++')
```

Nemeň mená metód a parametrov.

Tvoj odovzdaný program s menom `riesenie.py` musí začínať tromi riadkami komentárov:

```
# 11. zadanie: pajton
# autor: Janko Hraško
```

```
# datum: 24.11.2021
```

Projekt `riesenie.py` odovzdaj na úlohový server https://list.fmph.uniba.sk/ najneskôr **7. januára 2022** do 23:00. Môžeš zaň získať **5 bodov**.