Python 3 typy, funkce, kolekce MMXIV Jiří Fišer

1. Logické hodnoty

1) objekty třídy *bool* = konstanty: `True`, `False`
2) objekty jiných tříd jsou ve většině případů *pravdivé* (= *True*)
výjimky: *None*, čísla rovná nule, prázdné kolekce (i prázdný řetězec)
objekty pro něž vrací speciální metoda *__bool__* hodnotu *False* OR metoda *__len__* vrací hodnotu 0.
explicitní přetypování: **konstruktor** `bool(object)`
logické operátory: `and`, `or`, `not`

2. Číselné třídy (datové typy)

int

celá čísla s neomezeným rozsahem (*sys.maxsize* je jen největší číslo uložitelné do registrů procesoru)
Literály: `12`, `0xC`, `0o14` (oktalový), `0b1100` (binární)
Konstruktor: `int(x=0)`, `int(s, base=10)`

float

číslo v pohyblivé řádové čárce v dvojité přesnosti (=8 bytů)
Literály: 12.0, 1.2e1, 12e0
Konstruktor: `float(x)`, `float(s)`
speciální případy: `float("inf")`, `float("nan")`
funkce (*sqrt*, *sin*, *log*, apod.) : modul **math**

komplexní čísla

Literál: `12j`, `2+3j`, `1j` (nikoliv *j*)
Konstruktor: `complex(x,y=0)`, `complex(s)`
Operace: viz výše kromě operací využívajících zbytek
`c.real` (reálná část), `c.imag` (imaginární část)
`c.conjugate()`, `cmath.phase(c)` → argument
`cmath.polar(c)` →dvojice (r, argument)
`cmath.rect(r, argument)` → komplexní číslo *c*
další funkce modul **cmath**

decimal

číslo v pohyblivé řádové čárce s **dekadickou representací** a nastavitelnou přesností
Konstruktor: `decimal.Decimal(x)`, `decimal.Decimal(s)`

```
from decimal import *
getcontext().prec = 28 # globální nastavení přesnosti
Decimal(1) / Decimal(7)
#>>> Decimal('0.1428571428571429')
Decimal('3.14') #přesně
#>>> Decimal('3.14')
Decimal(3.14) # nejbližší representace float
#>>> Decimal('3.140000000000000012434...')
Decimal(2).sqrt() #podobně exp, ln, log10
#>>> Decimal('1.414213562373095048800...')
```

Lokální nastavení kontextu (přesnost, typ zaokrouhlení, &c)

```
with localcontext() as ctx:
    ctx.prec = 42   # vyšší přesnost
    print(Decimal(2).ln())
# normální přesnost
print(Decimal(2).ln())
```

fractions

zlomek (= racionální číslo), čitatel a jmenovatel je číslo *int*
Konstruktor: `fractions.Fraction(citatel=0, jmenovatel=1)`
`fractions.Fraction(float|dec|frac)`, `fractions.Fraction(s)`

```
Fraction(10, -4) #>>> Fraction(-5, 2)
Fraction(123)    #>> Fraction(123, 1)
Fraction("1/2")  #>> Fraction(1,2)
Fraction(Decimal('1.1'))
#>> Fraction(11,10)
Fraction(1.1)
#>> Fraction(2476979795053773, 2251799813685248)
```

Po konverzi z float čísel je možné využít metodu:
`limit_denominator(max_jmenovatel=1000000)`

```
Fraction(1,1).limit_denominator()
#>> Fraction(11,10) = nejbliž. zlomek se jmenov. < 1e6
Fraction(math.pi).limit_denominator(100)
#>> Fraction(311, 99)
```

Atributy: `f.nominator`, `f.denominator`
Operace: +,-,*,/, //, %, `math.floor(f)`, `math.ceil(f)`, `round(f)`

dělení apod.

`x / y` reálné dělení (bez ohledu na typ)
`x // y` celočíselné dělení resp. `floor(x/y)`
`x % y` zbytek po dělení = `y * (x // y)`
`abs(x)`, `x ** y` = `pow(x, y)`, `divmod(x,y)` → dvojice (x//y, x%y)

3. Řetězce

Unicode řetězce s maximální délkou *sys.maxsize* (a podpora nejen BMP)
Konstruktor: `str(obj)` # volá metodu *__str__* objektu
Literály: `'Tyr'`, `"Tyr"`, `"""dlouhý řetězec"""`, `'''dlouhý řetězec'''`
Dlouhé řetězce mohou být víceřádkové, a mohou obsahovat znaky ' a ".
`r'\syrový \string'` (a další kombinace) lomítka neuvozují esc. sekvence
Escape sekvence: `\n`, `\t`, `\'`, `\"` (a další jako v C), `\N{unicode-name}`, `\uXXXX` (16-bit unicode), `\UXXXXXXXX` (32-bit unicode)
Metody nad řetězci (*str*-objekty jsou neměnné!) :
`s.capitalize()` → první písmeno ve všech slovech velké
`s.center(width, fillChar=" ")` → řetězec symetricky doplněný vlevo i vpravo znaky *fillChar* na délku *width*
`s.count(subs)` → počet opakování *subs* v *s*
`s.endswith(subs)` → je *subs* na konci *s*?
`s.find(subs)` →pozice (index) *subs* v *s* nebo -1
`s.format(fmt, *args, **kwargs)` viz formátování
`s.index(subs)` → pozice (index) *subs* v *s* nebo výjimka *ValueError*
`s.isalnum()`, `s.isalpha()`, `s.isdigit()`, `s.isidentifier()`, `s.islower()`,, `s.isprintable()`, `s.isspace()`, `s.isuper()`
→ patří všechny znaky do dané unicode kategorie?
`s.join(iterable)` → řetězec spojující dohromady řetězce poskytované *iterable* (oddělovačem je *s*)

```
":".join(str(i) for i in range(1,3)) #>> '1:2'
```

`str.ljust(width, fillchar=" ")`
→ řetězec zleva doplněný znaky *fillChar* na délku *width*
`s.lower()` → řetězec, kde jsou velká písmena změněna na malá
`s.lstrip(chars=None)` → řetězec zleva zkrácený o jakékoliv znaky z řetězce *chars* (implicitně o mezerové znaky)
`s.partition(sep)` →trojice *(prefix,sep, postfix)* vzniklá rozdělením řetězce podle prvního výskytu *sub.*
`s.replace(old, new, count=∞)` → řetězec, v němž jsou zleva výskyty podřetězce *old* nahrazeny za *new* (maximálně *count*-krát)
`s.rfind`, `s.rindex`, `s.rpartition`, `s.rsplit`
obdoby metod *find, index, partition, split* s procházením zprava
`s.rstrip(chars=None)` → řetězec zprava zkrácený o jakékoliv znaky z řetězce *chars* (implicitně o mezerové znaky)
`s.split(sep, maxsplit=-1)` →seznam podřetězců-segmentů oddělených *sep.* Je-li *maxsplit*>1, pak je odděleno jen *maxpslit* položek.

```
'1,2,3'.split(',')
#>> ['1', '2', '3']
'1,2,3'.split(',', maxsplit=1)
#>> ['1', '2,3']
''.split(',') #není-li sep vrací seznam s původním 's'
#>> ['']
```

`s.split(maxsplit=-1)` seznam podřetězců-segmentů oddělených sekvencemi mezerových znaků.
`s.splitlines()` → podobné split, kde oddělovačem jsou separátory řádků ("\n", &c.)

```
''.splitlines() # rozdíl oproti 'split'
#>> []
```

`s.startswith(subs)` → je *subs* na začátku *s*?
`s.strip(chars=None)` → řetězec zleva i zprava zkrácený o jakékoliv znaky z řetězce *chars* (implicitně o mezerové znaky)
`s.upper()` → řetězec. kde jsou malá písmena změněna na velká
Operace nad řetězci (viz rozhraní read-only sekvencí)
řetězec = sekvence jednoznakových podřetězců
`len(s)` →délka řetězce
`s1 + s2` spojení (konkatenace) řetězců
pro postupné napojování řetězců, je výhodnější použít *join*

```
"".join("0123456789"[d] for d in digit_generator())
```

`s * i` je *i*-násobné opakování řetězce

```
"Programujte!" * 100

"o" in "Frodo"    #>> true
"oda" in "Frodo"  #>> false
"Frodo"[0]        # indexace od 0 >> "F"
"Frodo"[0:2]      # řez >> "Fr"
for z in "aeiouy": print(z) # string je iterable
```

4. Proměnné

Proměnné v *Pythonu* obsahují **odkaz** na objekty (bez určení jeho typu) a jsou **definovány prvním přiřazením**.

```
i = 15    # přiřazení a definice
i = "Tyr" # změna proměnné (odkazuje na str-objekt)
i = None
```

Hodnota *None* (singleton třídy *NoneType*) representuje nepřítomnost odkazu na objekt. *None* podporuje jen odkazové porovnání (identičnost):

```
x is None  #>> true, je-li v 'x' je odkaz na None
x is not None # negace předchozího
```

proměnné na úrovni modulu — jsou přímo viditelné ve svém modulu, po importu i v ostatních modelech

```
# modul testvar.py
p = 0
# jiný modul
import testvar
print(testvar.p) # s kvalifikací modulu
# nebo
from testvar import p
print(p) # zkopírování do jm. prostoru modulu
```

v těle funkcí je možno proměnné akt. modulu využívat přímo. Mohou však být nechtěně překryty lokální proměnnou (řešení: deklarace *global*)

```
x,y,z = 0,0,0
def f():
  print(z)  # odkaz na globální proměnnou
  x = 2 # definice nové lokální proměnné (překrytí)
  global y
  y = 2 # přiřazení do globální proměnné
```

atributy třídy fungují jako třídní (statické) proměnné. Musí být kvalifikovány jménem třídy (včetně metod dané třídy)

```
class X:
  p = 0
print(X.p) # kvalifikace jménem třídy
```

atributy instancí: nejčastěji definovány v konstruktoru. Musí být kvalifikovány instancí (v metodách třídy pomocí *self*)
lokální proměnné: definovány v těle funkcí (tvoří oblast viditelnosti). Viditelné i v těle vnořených funkcí (**lexikální vazba**). Mohou být výjimečně překryty (lze tomu zabránit pomocí deklarace *nonlocal*)
parametry funkcí: viditelnost stejná jako u lokálních prom.. Implicitní hodnoty **vyhodnoceny v okamžiku definice** funkce (nikoliv při volání!)

```
def f(s=[]):
  s.append(0)
  return s
f()  # vrací [0]
f()  # podruhé vrací [0,0]
```

5. Funkce

V Pythonu to jsou *entity prvního řádu* (lze vytvářet v libovolném místě, pracovat s nimi jako s jinými objekty).
Předávání hodnot:

```
def f(x,y,z=0): # 'z' s implicitní hodnotou
  return (x, y, z) # vrácena n-tice
f(1,2,3)  #>> (1,2,3) = poziční předávání
f(1,2)    #>> (1,2,0)
f(x=1, y=2, z=2) #>> (1,2,2) = předávání jménem
f(1, y=5) #>> (1,2,5)= smíšené: pozičně, zbytek jménem
```

Parametry předávané povinně jménem (po hvězdičce)

```
def f(x=0, *, y, z=1):
  # x pozičně nebo jménem (resp. lze vynechat)
    # y jen jménem (nelze vynechat!)
    # z jen jménem (lze vynechat)
f(0, y=2)
```

Nespecifikované poziční parametry:

```
def f(*args): # args je sekvence nespecif. parametrů
  return args[0]
f(1,2,3) #>> 1
# namísto jedn. parametrů lze předat seznam
pars = [1,2,3]
f(*pars) # prefix hvězdička
```

Nespecifikované pojmenované parametry:

```
def f(**kwargs): # kwrags je slovník pojmen. parametrů
  return list(kwargs.keys())
f(x=1, y=2) #>> ["x", "y"]
# namísto jedn. parametrů lze předat slovník
npars = {"x" : 1, "y" : 2}
f(**npars)  # prefix: dvě hvězdičky
```

Parametry lze kombinovat v pořadí: *specifikované*, *nespecifikované poziční* (**args*) a *nespecifikované pojmenované* (***kwargs*).

Lambda funkce

literály anonymních funkcí tvořených jen jedním výrazem.

```
lambda x : x + 1→ funkce s jedním parametrem
lambda : print("ahoj")
```

→ bezparametrická funkce (s odloženým prováděním)

6. Příkazy

Přiřazovací příkaz:

```
x = 2   # jednoduché přiřazení
x += 2  # složené přiřazené
(x,y) = (1,2) # per partes přiřazení
x, y = 1, 2   # lze vynechat závorky
x, y = y,x    # výměna pomocí per partes
x, *p = 1,2,3 # x = 1, p = [2,3] per partes se zbytkem
```

Větvení:

```
if podm1:
  # je-li podm1 = true
[elif podm2:
  # je-li podm1 = false and podm2 = true]
...
[else: # jsou-li všechny podmínky nepravdivé]
```

Cyklus *while*:

```
while podm:
  # tělo cyklu
[else: # je-li cyklus opuštěn nesplněním podmínky]
```

Iterační cyklus:

Cyklus iteruje přes **všechny prvky iterátoru** (je získán voláním `iter(iterable)`) a přiřazuje je do proměnné *var.*

```
for var in iteratorable:
  # tělo cyklu
[else: # je-li cyklus ukončen vyčerpáním iterátoru]
```

Typické příklady:

```
for i in range(1,10):
  # i nabývá hodnot 1 až 9 (včetně)
for key in slovnik.keys():
  # iterace přes klíče slovníku
for key, value slovnik.items():
  # iterace přes dvojici (klíč, hodnota)
for i, item in enumerate(iterable):
  # iterace přes dvojici (index, prvek)
```

Příkazy skoku: *return*, *break*, *continue*

Příkaz aserce:

```
assert podm, string
```

Je-li podmínka **nepravdivá**, je vyvolána výjimka *AssertionError* s řetězcem jako zprávou (je-li zapnuta optimalizace, pak příkaz nic nedělá).

```
assert x>=0, "Záporná hodnota parametru!"
```

Prázdný příkaz (pass):
Užívá se u deklarací funkcí a tříd s prázdným tělem. Obecně lze využít na místě libovolného příkazu (nic nevykonává).

```
def f(x): pass
class Simple: pass
```

7. Iterovatelné objekty a iterátory

Iterovatelné objekty poskytují **standardní iterátory**, které umožňují iteraci přes všechny jejich prvky.
Získání standardního iterátoru (nezávislé instance):

```
iterator = iter(iterable)
```

Iterátor je zároveň *iterovatelným objektem* (je sám sobě iterátorem)
Posun iterátoru (vrací další prvek):

```
item = next(iterator)
```

Po dosažení konce je vyvolána výjimka *StopIteration.*

Poskytovatelé iterátorů

`range(min, max, k)` → iterátor od *min* (včetně) po *max* (vyjma) krok *k*
`itertools.repeat(elem, n= ∞)` → iterátor *n*-krát vracející objekt *o*
`iter(func, zarazka)`
→ iterátor hodnot vrácených opakovaným voláním funkce *f*, dokud je hodnota ≠*zarazka*

Konzumenti iterátorů

Iterovatelné objekty lze využívat v cyklu *for* a v dalších **metodách**:
`min(iterable, *, key, default)`,`max(iterable, *, key, default)`
→ minimální/maximální prvek z iterátoru (*default*, je-li prázdný),
key funkce definující hodnotu pro uspořádání (viz *sorted*)
`all(iterable)`
→ *true*, jsou-li všechny položky iterátoru pravdivé
`any(iterable)`
→ *true*, je-li alespoň jedna položka pravdivá
`sorted(iterable, *, key, reverse)`
→ seznam vzestupně setříděných prvků (stabilní třídění)
key funkce definující hodnotu pro uspořádání (implicitně identita)
reverse setřídění v opačném gardu (sestupně)

```
p=[2+3j, 5+7j, 2-1j, 3]
sorted(p) # nefunguje, kompl. čísla nemají uspořádání
# TypeError: unorderable types: complex() < complex()
sorted(p, key=lambda c: c.real)
#>> [(2+3j), (2-1j), 3, (5+7j)]
```

`functools.reduce(func, iterable, initializer=None)`→ hodnota získaná aplikací binární funkce *func* zleva doprava mezi položkami iterátoru (initializer=hodnota přidaná zleva).

```
reduce(operators.add, range(2,4), 1) #>> 1+2+3
```

Filtry iterátorů

přijímají iterátor a vrací nový modifikovaný
`map(func, iterableI)`
→ iterátor *func(i1), func(i2)*, …
`filter(func, iterableI)`
→iterátor, těch položek z *iterable*, pro něž *func(i)=True.*
Mapování a filtrování lze zapsat i pomocí **iteračové komprehenze**:

```
map(mapfunc, filter(pred, iterable))
# lze zapsat jako
(mapfunc(o) for o in iterable if pred(o))
# praktický příklad
(abs(x) for x in range(-5, 5) if x % 2 == 0)
#>> iterátor 4,2,0,2,4
```

`zip(iterableA, iterableB, …)`
→ iterátor n-tic (a1,b1…), (a2, b2…), … (podle kratšího iterátoru)
idiom pro **unzip**

```
x = [1, 2, 3]; y = [4, 5, 6]
zipped = zip(x, y)
#>> [(1, 4), (2, 5), (3, 6)]
x2, y2 = zip(*zipped)
```

`enumerate(iterableI, start=0)`
→ vrací iterátor dvojic index, prvek tj. *(start, i1), (start+1, i2)*, …

```
for i,p from enumerate(iterable):
  print("index {0}, prvek {1}".format(i,p))
```

Další iteračové filtry jsou v modulu *itertools.*

```
cycle("ABC") #>> A,B,C,A,B,C,A,B,...
chain("ABC","DEF","G") #>> A,B,C,D,E,F,G
compress("ABCDEF", [1,0,1,0,1,1]) #>> A,C,E,F
dropwhile(lambda x: x<5, [1,4,6,4,1]) #>> 6,4,1
islice('ABCDEFG', 2, 4) #>> C,D
takewhile(lambda x: x<5, [1,4,6,4,1]) #>> 1,4
product('ABC', 'ABC') #>> AA AB AC BA BB BC CA CB CC
permutations('ABC', 2) #>> AB AC BA BC CA CB
combinations('ABC', 2) #>> AB AC BC
combinations_with_replacement('ABC', 2)
#>> AA AB AC BB BC CC
```

8. Sekvence

Sekvence = **iterovatelné** kolekce podporující poziční indexaci.
Metody a operace společné všem sekvencím:
abstraktní bázový typ: `collections.abc.Sequence`
`o in s` → je objekt *o* obsažen v sekvenci *s*
`o not in s` → *true*, není-li objekt *o* obsažen v *s*
`s1 + s2` →zřetězení sekvencí
`s * n` →sekvence vzniklá *n*-násobným opakováním *s*
`s[i]` →položka s indexem *i*
`s[i:j:k]` →řez = podsekvence s indexy [i,j)
`len(s)` → délka sekvence
`s.index(o)` → pozice položky *o* v sekvenci *s*, nebo -1
`s.count(o)` → počet výskytů *o* v sekvenci *s*
Metody měnitelných sekvencí:
abstraktní bázový typ: `collections.abc.MutableSequence`
`s[i] = x` změna položky (= přesměrování odkazu)
`s[i:j] = iterable` nahrazení podsekvence položkami z *iterable*
`del s[i]`,`del s[i:j]` výmaz položky (položek)
`s.append(o)` přidání položky na konec
`s.clear()` výmaz všech položek (= `del s[:]`)
`s.copy()` → mělká kopie položek (= `s[:]`)
`s.extend(iterable)` přidání položek z iterable na konec
`s.insert(i, o)` vložení položky na index *i* (+posun)
`s.pop(i=0)` → položka vyňatá z indexu *i*
`s.remove(o)` vyjme první výskyt položky o
`s.reverse()` obrací sekvenci

Seznam (list)

Seznam je hlavní implementací měnitelné sekvence.
Literál: `[2, "a", 5.0]`, `[]` (prázdný seznam)
Konstruktor: `list(iterable)` — vytváří nový seznam z prvků iterátoru

```
s = list(range(2,100,2)) # seznam čísel 2,4,6,...98
```

Komprehenze: `[f(x) for x in iterable if pred(x)]`
Další měnitelné sekvence:
`collections.dequeu` = optimalizována pro manipulaci (přidání, výmaz) na obou koncích

N-tice (tuple)

N-tice je hlavní representací neměnné sekvence (po vytvoření nelze přidávat či odebírat prvky).
Literál: `(1, "a")`, `(1,)` (jednoprvková n-tice), závorky lze často vynechat
Konstruktor: `tuple(iterable)`
využití:
A) dočasné seskupení hodnot (přepravka)
B) (nehomogenní) záznam s pozičně identifikovanými položkami.

9. Množiny

Kolekce neuspořádaných jedinečných prvků.
abstraktní bázový typ: `collections.abc.Set` (resp. `.MutableSet`)
Operace viz sekvence kromě metod využívajících indexy.
Množinové operace:
`s1.union(s2,…)` → sjednocení (také `s1 | s2`).
`s1.intersection(s2,…)` → průnik (také `s1 & s2`)
`s1.difference(s2,…)` → rozdíl (také `s1 - s2`)
`s1.symmetric_difference(s2)` → symetrický rozdíl (`s1 ^ s2`)
`s.isdisjoint(s2)` →jsou disjunktní?
`s.issubset(s2)` → je *s1* podmnožinou *s2* (také `s1<=s2`, podobně `s1<s2`)
`s.issuperset(s2)` → je *s1* nadmnož. *s2* (také `s1>=s2`, podobně `s1>s2`)

Množina měnitelná

representace pomocí *hashovací tabulky*
Literál: `{2, "a", 5.0}`, `set()` (prázdná množina)
Konstruktor: `set(iterable)` — vytváří nový seznam z prvků iterátoru

```
s = set(range(2,100,2)) # množina čísel 2,4,6,...98
```

Komprehenze:

```
{f(x) for x in iterable if pred(x)}
```

`s.discard(o)` odstraní prvek *o*, je-li přítomen
(`s.remove(o)` vyvolá *KeyError*, je-li nepřítomen)
Neměnitelná množina: `frozenset(iterable)`

10. Zobrazení (mapování)

Representují zobrazení množiny klíčů do množiny hodnot.
Operace a metody:
`len(d)` → počet dvojic (klíč, hodnota)
`d[key]` zkratka za d.get(key)) → hodnota pro klíč key
`d[key] = value` nastavení (přepsání) hodnoty pro klíč
`del d[key]` výmaz dvojice (klíč, hodnota)
`key in d` → je daný klíč obsažen
`d.clear()` výmaz celého slovníku
`d.copy()` → mělká kopie
`d.get(key, default)`→ hodnota pro daný klíč nebo *default* (není-li klíč)
`d.popitem()` → vrací a vyjímá náhodně jednu z dvojic (klíč, hodnota)
`d.setdefault(key, default)`
→ hodnota pro daný klíč nebo *default* + nastavení (není-li klíč)
`d.update(mapping)` rozšíří mapování o klíče z jiného
`d.keys()` → iterovatelná kolekce klíčů (optimaliz. pro hledání)
`d.values()` → iterovatelná kolekce hodnot
`d.items()` → iterovatelná kolekce dvojic (klíč, hodnota)

Slovník

Základní implementace zobrazení pomocí hashovací tabulky.
Objekty klíčů musí být hashovatelné (= implementující metodu *__hash__*).
Literál: `{``a`` : 2, "b" : 3, "c" : 5}`,`{}` (prázdný slovník)
Konstruktor: `dict(mapping)`, `dict(iterable)` (přes seznam dvojic)
`dict(**kwargs)` = zobrazení identifikátorů parametrů na parametry

```
d = dict(a=2, b=3, c=5)
```

Komprehenze:

```
{key : value for key, value in iterable}
# jsou možné i další typy
{x : f(x) for x in iterable}
```

Další druhy slovníků (v modulu *collection*):
`OrderedDict` s uspořádanými klíči (podle pořadí vložení)
`defaultdict` vrací defaultní hodnotu pro neexistující klíč
`ChainMap` spojení více zobrazení do jediného (virtuálního)
`Counter` čítač výskytů klíčů (hodnoty jsou automaticky inkrementovány)