Python I — čísla a kolekce

int

celá čísla (*integer*) = $\mathbb{Z}$, rozsah dán jen pamětí

```
i = 42
i = 0b10_1010 # binary
i = 0o52 # octal
i = 0x2a # hex
i = int("42") # from string
i = int("52", base=8)
i = int(42.78) # from float

bi = 2_000_000_000 # 2e9 is float!
```

Převod na řetězec

`str(i)` → `'42'`

`f"{i:3}"` → `' 42'`: šíře 3, výplň mezery vlevo

`f"{i:03}"` → `'042'`: šíře 3, výplň nuly vlevo

`f"{i:04X}"` → `'002A'`: headecimální (výplň nuly)

`f"{i:08b}"` → `'00101010'`: binární (výplň nuly)

`f"{1001:,}"` → `'1,001'`: oddělovač skupin (*locales*)

`f"{2_022:07,}"` → `'002,022'`

Operace

`i+1` → `43`: sčítání (*addition*)

`i+1.0` → `43.0` [float]: impl. přetypování (*default cast*)

`-i` → `-42`: unární minus, opačná hodnota (*negative*)

`i*2` → `84`: násobení (*multiplication*)

`i / 5` → `8.4` [float]: reálné dělení (*real division*)

`i // 5` → `8` [int]: celočíselné dělení (*integer/floor div.*)

`i % 5` → `2` [int]: zbytek po dělení, modulo (*modulus*)
T: `a % b == a - b * (a // b)`

`i**2` → `1764`: umocňování (*exponentation*)

`i << 2` → `168`: bitový posun vlevo (*left shift*)
T: `a << s == a * (2**s)`

`i >> 2` → `10`: bitový posun vpravo (*right shift*)
T: `a >> s == a // (2**s)` (znaménkový posun)

a	0	0	1	1	3	
b	0	1	0	1	5	
`a | b`	0	1	1	1	7	bitwise OR
`a & b`	0	0	0	1	1	bitwise AND
`a ^ b`	0	1	1	0	6	bitwise XOR
`~a`	1	1	0	0	-4	bitwise negation
`a << 1`	0	1	1	0	6	left shift
`a >> 1`	0	0	0	1	1	right shift

čísla s nejvyšším bitem 1 jsou interpretována jako záporná čísla: dvojkový doplněk, (*two's complement*).
T: `-a = ~a + 1`

Funkce

`math.factorial(10)` → `2432902008176640000`

`math.comb(10, 5)` → `252`: binom. koeficient $\binom{10}{5}$

`math.gcd(144, 120)` → `24`: největší společný dělitel

`math.isqrt(50)` → `7`: celočíselná odmocnina 3.8

float

pohyblivá řádová čárka $\subset \mathbb{Q}$
(*floating point*): $m \cdot 2^e$

$m \in [1, 2)$ (mantisa), $e \in -1022 \ldots + 1023$ (exponent)

64 bitů (52 mantisa, 1 znaménko, 11 exponent)

přesnost: cca 15 platných dekadických číslic

rozsah:

$(-1.8 \times 10^{308}, -2.2 \times 10^{-308})$	záporná
$\{-0.0, 0.0\}$	nuly (i záporná)
$(2.2 \times 10^{-308}, 1.8 \times 10^{308})$	kladná
$\{-\infty, +\infty, \text{NAN}\}$	speciální hodnoty

```
x = 3.14
x = float("3.14")

sx = -0.008
sx = -8e-3 # -8 . 10^-3
bx = 2_000_000.0 # decimal point mandatory
bx = 2e6

nan = math.nan # not-a-number
inf = math.inf # infinity
```

převod na řetězec

`str(bx)` → `'2000000.0'`

`f"{x:6.3}"` → `'  3.14'`
univerzální (fixní pro běžné řády)

`f"{bx:6.3}"` → `' 2e+06'`
univerzální (exponenciální pro vyšší řády)

`f"{x:6.3f}"` → `' 3.140'`: fixní (bez exponentu)

`f"{x:10.3e}"` → `' 3.140e+00'`
exponencialní (semilogaritmická)

operace

základní operace (+,-,*,/) jsou obdobou celočíselných jejich výsledky však mohou být nepřesné.

`0.2 + 0.2` → `0.4`

`0.2 + 0.1` → `0.30000000000000004` (0.3)

celočíselné dělení = $\left\lfloor \frac{a}{b} \right\rfloor$! občas podivné výsledky

`11.0 // 2.2` → `4` (správně 5)

`11.0 % 2.2` → `2.1999999999999993` (0)

funkce

`abs(-x)` → `3.14`

`math.atan2(1, 2)` → `0.463647609`
`atan2(y, x)` = $\arctan(y/x)$ vracející úhel $[0, 2\pi)$

`math.ceil(x)` → `4` [int]

`math.floor(x)` → `3` [int]

`math.isclose(3 * 0.1, 0.3)` → `True`
relativní chyba $< 10^{-9}$

`math.isclose(math.sin(math.pi), 0, abs_tol=1e-9)` → `True`
absolutní chyba $< 10^{-9}$ (vhodné pro okolí 0)

`math.isnan(math.nan)` → `True`
pozor! `math.nan == math.nan` → `False`

`math.sin(math.pi)` → `1.2246468e-16` (0)
podobně i `cos`, `tan`

`math.exp(x)` → `23.10386686`

`math.log(x)` → `1.1442228`: přirozený logaritmus

`math.log(x,2)` → `1.65076456`
binární logaritmus (spec. funkce `log2`)

`math.log(x,10)` → `0.496929648`
dekadický logaritmus (spec. funkce `log10`)

`math.sqrt(x)` → `1.772`: odmocnina (= `x ** 0.5`)

string (řetězec)

posloupnost znaků ze znakové sady Unicode

```
# two types of separators
\textsc{s = 'Minas\nTirith'}
# multiline string literal (long)
s = """Minas
Tirith"""

#special interpretation
sraw = r"Minas\nTirith" # raw string
# \n is not new line but two chars
sf = f"pi={math.pi}" #interpolation str.
#other (combinated) literals are possible
srf = '''\pi=
{math.pi}''' #raw long with interpolation

es = "" # empty string
unstr = "\t  Gondor \n  "
```

indexování a výřezy

`s[0]` → `"M"` [string]: substring of first character

`s[1]` → `"i"`: second character

`s[-1]` → `"h"`:last character

`s[1:3]` → `"in"`
výřez (*slice*) od index 1 (včetně) do 3 (vyjma)
obsahující 3-1 = 2 znaky

`s[6:-1]` → `'Tirit'`
výřez od indexu 6 (včetně) do posledního (vyjma)

`s[1:]` → `'inas\nTirith'`
výřez od druhého znaku do konce (bez prvního)

`s[:-1]` → `'Minas\nTirit'`
bez posledního znaku (ekvivalentní s `s[0:-1]`)

`s[1:-1:2]` → `'ia\nii'`
každý sudý znak (bez prvního a posledního)

`s[::-1]` → `'htiriT\nsaniM'`: reversed string

`s[-1:1:-1]` → `'htiriT\nsan'`
od posledního (včetně) pro druhý (index 1) vyjma

operátory a funkce

`"as" in s` → `True`

`s + "!"` → `'Minas\nTirith!'`
zřetězení (*concatenation*)

`s * 2` → `'Minas\nTirithMinas\nTirith'`

`len(s)` → `12`: počet znaků

`ord("o")` → `111`: unicode pozice (prvního) znaku

`chr(111)` → `"o"`: znak na unicode pozici

metody

`s.count("i")` → `3`
počet nepřekrývajících se výskytů podřetězce

`s.endswidth("th")` → `True`

`s.startswith("M")` → `True`

`s.find("i")` → `1`, `s.index("i")` → `1`
vrací index prvního výskytu podřetězce
(není-li obsažen: `find` → `-1`, `index` výjimka)

`s.rfind("i")` → `9`, `s.rindex("i")` → `9`
obdoba pouze se vrací poslední výskyt

`"|".join(["Minas","Anor"])` → `'Minas|Anor'`
spojení seznamu řetězců (s vloženým oddělovačem)

`s.ljust(16)` → `'Minas\nTirith     '`

`s.rjust(16, "x")` → `'xxxxMinas\nTirith'`
výplň na požadovanou šířku (vlevo, vpravo)

`s.lower()` → `'minas\ntirith'`
převod na malá písmena (minusky)

`s.upper()` → `'MINAS\nTIRITH'`
převod na velká písmena (verzálky)

`unstr.lstrip()` → `'Gondor \n  '`

`unstr.rstrip()` → `'\t  Gondor'`

`unstr.strip()` → `'Gondor'`
odstranění mezer. znaků zleva/zprava/z obou stran

`s.partition("\n")` → `('Minas','\n','Tirith')`
vrací trojici (prefix, separátor, suffix)

`s.removeprefix("Minas")` → `"\nTirith"` 3.9
odstraňuje prefix (pokud jím řetězec začíná)

`s.replace("Tirith", "T.¨")` → `'Minas\nT.'`
vrací řetězec se substitucí podřetězce

`s.split("\n")` → `['Minas', 'Tirith']`
rozděluje podle oddělovače

`s.splitlines()` → `['Minas', 'Tirith']`
vhodnější pro dělení řádků

tuple (n-tice)

neměnná sekvence, s (pozičně) indexovatelnými prvky

```
t = (10, 11, 12)
t = 10, 11, 12 # without paranthesis
t = tuple(range(10, 13))

t1 = (42, ) # tuple with one item
t0 = () # tuple with zero items
nt = (("a", "b"), 10) # nested tuple
```

operátory a metody

`t[0]` → `10`

`nt[0][1]` → `"b"`

`x, y, z = t`: dekonstrukce)
x←10, y←11, z←12

`x, *r = t`: částečná dekonstrukce
x←10, r←[11, 12] [list]

`len(t1)` → `1`: (fixní) délka ntice

`10 in t` → `True`: je prvkem

list (seznam)

modifikovatelná sekvence s indexovatelnými prvky

```
l = [10, 11, 12, 13]
l = list(range(10, 14))

el = () # empty list
nt = [["a", "b"], [1, 2]] # nested list
```

komprehenze

`[2*i for i in l]` → `[20, 22, 24, 26]`

`[i//2 for i in l if i%2 == 0]` → `[5, 6]`

indexování a výřezy

`l[1]` → `11`: $O(1)$

`l[1:2]` → `[11]`: $O(k)$ (k velikost výřezu)

`l[2::-1]` → `[12, 11, 10]`

použití je stejné jako u řetězců, lze je však využívat i pro změnu položek

`l[0] = 0` ⟳ `[0, 11, 12, 13]`: mění první prvek

`l[1:3] = [0, 1]` ⟳ `[10, 0, 1, 13]`

`l[1:3] = []` ⟳ `[10, 13]` : výmaz více prvků

`l[1:-1] = l[-2:0:-1]` ⟳ `[10, 12, 11, 13]`
obrácení vnitřních prvků

operátory / příkazy

`l + [100, 200]` → `[10, 12, 11, 13, 100, 200]`
zřetězení seznamů

`l * 2` → `[10, 11, 12, 13, 10, 11, 12, 13]`

`12 in l` → `True`: $O(n)$

`del l[0]` ⟳ `[11, 12, 13]`

`del l[:2]` ⟳ `[12, 13]`
alternativní způsob vyjmutí prvků

funkce a metody

`min(l)` → `10`: $O(n)$, podobně `max`

`l.index(12)` → `2` $O(n)$¨

`l.count(10)` → `1`: $O(n)$

`l.append(0)` ⟳ `[10, 11, 12, 13, 0]`: $O(1)$

`l.extends([0, 1])` ⟳ `[10, 11, 12, 13, 0, 1]`
$O(n_2)$

`l.insert(0, 1)` ⟳ `[1, 10, 11, 12, 13]`: $O(n)$

`l.clear()` ⟳ `[]`

`l.pop()` → `13` ⟳ `[10, 11, 12]` : $O(1)$

`l.pop(0)` → `10` ⟳ `[10, 11, 12, 13]` : $O(n)$

`l.remove(13)` ⟳ `[10, 11, 12]`: $O(n)$

`l.reverse` ⟳ `[13, 12, 11, 10]`: $O(n)$

dictionary (slovník)

```
d = {"a" : 1, "b" : 2}
# map key (e.g. "a") to value (1)
d = dict(a=1, n=2) #  only for string keys
ed = {} # empty dictionary
```

od 3.7 jsou klíče uloženy v pořadí vkládání

komprehenze

`{x: x+1 for x in range(2)}` → `{0: 1, 1:2}`

operátory

`d["a"]` → `1`: $O(1)$ najde hodnotu ke klíči

`d["a"] = 0` ⟳ `{"a" : 0, "b" : 2}`
změna hodnoty (existující klíč) $O(1)$

`d["c"] = 0` ⟳ `{"a" : 1, "b" : 2, "c": 0}`
přidání nového klíče s hodnotou $O(1)$

`"a" in d` → `True`: test existence klíče $O(1)$

`len(d)` → `2`: počet klíčů (= hodnot)

`d | {"c" : 0}` → `{"a" : 1, "b" : 2, "c": 0}`

`d |= {"c" : 0}` ⟳ `{"a" : 1, "b" : 2, "c": 0}`
až od verze 3.9 jinak `update`

metody

`d.get("a", None)` → `1`, `d.get("x", 0)` → `0`
u neexistujícího klíče vrací druhý parametr (impl. hodn.)

`d.items()` → `[("a", 1), ("b", 2)]` [view]

`d.keys()` → `["a", "b"]`: [view]

`d.values()` → `[1, 2]`: [view]

`view` je dynamický pohled (mění se při změně slovníku)

`d.clear()` ⟳ `{}`

`d.update({"c": 0})`
⟳ `{"a" : 1, "b" : 2, "c": 0}`

`d.pop("a", None)` → `1` ⟳ `{"b": 2}`

`d.pop("x", None)` → `None` ⟳ `{"a": 1, "b": 2}`

`d.setdefault("a", 0)` → `1`
⟳ `{"a": 1, "b": 2}` vrací hodnotu exist. klíče

`d.setdefault("x", 0)` → `0`
⟳ `{"a": 1, "b": 2, "x": 0}` přidává neexistující

set (množina)

nesupořádaná množina s rychlým vyhledáváním

```
s1 = {1, 2, 3}
s2 = {2, 5}
es = set() #empty set
```

komprehenze

`{x for x in range(2)}` → `{1, 2}`

operátory

`1 in s1` → `True`: $O(1)$

`s1 & s2` → `{2}`
průnik, také `s1.intersection(s2)`

`s1 | s2` → `{1, 2, 3, 5}`
sjednocení, také `s1.union(s2)`

`s1 - s2` → `{1, 3}`
množinový rozdíl, také `s1.difference(s2)`

`s1 ^ s2` → `{1, 3, 5}`
symetrický rozdíl, také `s1.difference(s2)`

`s1.isdisjoint(s2)` → `False`
je disjunktní (není průnik)

`s1 < s2` → `False`: vlastní podmnožina ($\subset$)

`s1 <= s2` → `False`: podmnožina ($\subseteq$)

`s1.add(4)` ⟳ `{1,2,3,4}`

`s1.update(s2)` ⟳ `{1, 2, 3, 5}`: připojí množinu
také `s1 |= s2`

`s1.remove(1)` ⟳ `{2, 3}`

`s1.pop()` ⇝ `{1, 3}`: vyjme náhodný prvek

Python II — protokoly, byty

logické hodnoty (bool)

práve dvě instance representující pravdivostní hodnoty

```
t = True
f = False
```

přetypování

`bool(0)` → `False`
`bool(0.0)` → `False`: nevhodné
`bool(-0.0)` → `False`: nevhodné
`bool(False)` → `False` (pro úplnost)
`bool("")` → `False`
`bool([])` → `False`
`bool({})` → `False`

podobně všechny prázdné kolekce (`len(c) == 0`)

ostatní objekty jsou interpertovány jako `True`.

přetypování lze v kontextu, kdy je očekávána logická hodnota (podmínky), vynechat:

```
x = 1
if x:  # better: bool(x), best: x != 0
  print("non zero")
```

operátory

a	False	False	True	True
b	False	True	False	True
`a or b`	False	True	True	True
`a and b`	False	False	False	True
`not a`	True	True	False	False

T: `not (a and b) == not a or not b`
T: `not (a or b) == not a and not b`

polymorfismus

kód (funkce) je polymorfní, pokud dokáže pracovat s objekty různých tříd

```
def max(a, b):
  return a if a > b else b
```

objekty v polymorfním kódu **musí sdílet** společné metody (operátory) se stejným kontraktem (chováním). v Pythonu **nepotřebují** sdílené (pod)rozhraní explicitně specifikovat (stačí jen form. shoda a dokumentace)

duck test: *If it walks like a duck and it quacks like a duck, then it must be a duck.* (and it don't have to be certified)

equatable (porovnatelné) objekty

objekty, u nichž lze určit zda jsou shodné či nikoliv

operátory: ==, !=

T: `not (a == b) == a != b`
T: `a = a` je vždy `True` (reflexivnost)
T: `a = b and b = a` je vždy `True`: symetričnost
T: pokud `a = b and b = c` pak `a = c`: tranzitivnost

`5 == 5` → `True`
`5.0 == 5.0` → `True`: lépe `a.isclose(b)`
`True == True` → `True`: de facto logická implikace
`"pes" == "pes"` → `True`
znaky na všech pozicích shodné (! alternativní repres.)
`[1, 2] == [1, 2]` → `True`
prvky na všech pozicích shodné (závisí na pořadí)
`{1, 2} == {2, 1}` → `True`
shodné množiny (stejné prvky bez ohledu na pořadí)
`{1: 2, 2: 3} == {2: 3, 1: 2}` → `True`
shodné dvojice (klíč, hodnota), nezávisí na pořadí

existuje i vztah **identity**: `a is b`
proměnné a a b odkazují na stejné objekty v paměti

hodnotové porovnání: shodné objekty nemusí být identické (stačí, když mají stejný obsah)
příklad: čísla, log. hodnoty, řetězce, kolekce
pokud jsou neměnné může být výhodné udržovat jen jednu instanci hodnoty (není povinné!)

odkazové porovnání: objekty jsou shodné, právě tehdy když jsou identické
příklad: objekty representující věci, lidi, GUI widgety

`x is None` efektivnější než `x == None` (odkaz. porov.)

orderable (uspořádatelné) objekty

objekty, u nichž existuje přirozené uspořádání

operátory: <, >, <=, >=

T: `not (a < b) == a >= b`
T: pokud `a <= b and b >= a` pak `a == b`
T: `a < b and b > a` je vždy `False`: antisymetričnost
T: pokud `a < b and b < c` pak `a < c`: tranzitivita

uspořádání je na rozdíl od shodnosti definované jen u některých tříd

čísla: `3 < 5` → `True`
log. hodnoty: `False < True` → `True`

lexikografické uspořádání pro sekvence: porovnávají se postupně prvky a první neshodný rozhodne.
`[1, 2, 3] < [1, 2, 5]` → `True` neboť 3 < 5
`(1, 2) < (1, 2, 0)` → `True` neboť [1, 2] je prefix
u řetězců se porovnávají Unicode pozice znaků viz `ord`
`"bok" < "buk"` → `True`
neboť `ord("o")` = 111 je menší než `ord("u")` = 117
`"orel" < "čáp"` → `True`: ‼ `ord("o")` = 269

řetězení operátorů uspořádání (specialita Pythonu):
`1 < 2 < 3` → `True`: totéž jako `1 < 2 and 2 < 3`

iterable (iterovatelné) objekt

objekty, poskytující na požádání iterátory

iterátor: objekt, který po každém volání `next` vrací další člen posloupnosti

základními iterovatelnými objekty jsou kolekce: poskytují (nezávislé) iterátory vracející postupně všechny prvky kolekce

```
c = [1, 2, 3] # list is iterable
it = iter(c) # get iterator over list
i2 = iter(c)
# get new (independent) iterator over list
```

`next(it)` → `1`: následující prvek iterátoru

po dosažení konce funkce `next` vyhazuje výjimku `StopIteration`.

Iterátor vrací posloupnost jen jednou (po vyhození výjimky na konci je vyčerpán). Lze však získat nový.

`list(it2)` → `[1, 2, 3]`: plní z iterátoru seznam

přes iterovatelné objekty (nikoliv iterátory) lze iterovat cyklem `foreach`

```
for i in c: # c is list = iterable
  print(l)
```

lenivé (lazy) iterátory

iterátory, které nejsou spojeny s kolekcemi a vyžadují alokaci paměti jen pro jeden – aktuální prvek, nikoliv všechny:

```
r = range(1, 3_000_000_000)
ri = iter(r) # iterator over range

f = open("file", "rt")
fi = iter(f) # iterator over lines of file
```

u lenivých iterátorů je často iterovatelný objekt zároveň (svým) iterátorem tj. platí `it = iter(it)`

```
l = [1, 2, 3] # iterable (no iterator)
l2 = [5, 6] # also iterable
cs = itertools.cycle(l) # lazy iterator
# iterable and also iterator over infinity
# sequence of 1, 2, 3, 1, 2, 3, 1, ...
```

kombinování a transformace iterátorů

parametrem je vždy iterovatelný objekt a výsledkem je lenivý iterátor (jenž je zároveň iterovatelným objektem)

`import itertools as itools`

`zip(l1, l2)`
→ iter[2]: `(1,5), (2,6)`

`itertools.zip_longest(l, l2)`
→ iter[3]: `(1,5), (2,6), (3,None)`

`itools.chain(l, l2)` → zřetězení iter.
iter[6]: `1, 2, 3, 5, 6`

`itools.product(l, repeat=2)` → kart. součin
iter[9]: `(1,1), (1,2), (1,3), (2,1), (2,2) ...`

`itools.permutations(l)` → permutace
iter[6]: `(1,2,3),(1,3,2),(2,1,3),(2,3,1), ...`

`itools.permutations(l, 2)`
→ 2-prvkové variace
iter[6]: `(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)`

`itools.combinations(l, 2)`
→ 2-prvkové kombinace bez opakování
iter[3]: `(1,2), (1,3), (2,3)`

`itools.combinations_with_replacement(l,2)`
→ 2-prvkové kombinace s opakováním
iter[6]:`(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)`

lenivost v praxi:

`p = itools.permutations(range(1,11))`
lenivý iterátor přes 3 628 800 permutací (nespočítány)

`ps = itools.islice(p, 90, 91)`
→ iter[1]:`(1,2,3,4,5,9,10,6,7,8)`
lenivý iterátor vracející 91. permutaci (nespočtena)

`list(ps)` → [(1, 2, 3, 4, 5, 9, 10, 6, 7, 8)]
90 permutací spočítáno a vyhozeno, 91. do seznamu

`next(p)` → `(1, 2, 3, 4, 5, 9, 10, 6, 8, 7)`
vrácena 92. permutace, 3628708 zbývá spočítat

generátor – iterátorová komprehenze

podobná seznamové komprehenzi, vrací ale lenivý iterátor (kulaté závorky jsou občas nepovinné)

`(str(i) for i in itools.cycle([1,2]))`
→ iter[∞]: "1", "2", "1", "2", ...

`sum(x for x in range(8000))` → `31996000`
suma čísel od 0 do 7999 (extrémně neefektivní)

bytes – pole bytů

specializovaná **neměnná** kolekce pro úsporné ukládání bytů (hodnot 0-255).

lze ji také chápat jako řetězec jednobytových (extended) ASCII znaků, viz níže literály s prefixem b.

```
b = b'\x00\x2e\x3f\x85'
# hexadecimal escape sequences
b = b'\x00.?\x85'
# values 32—127 are representable
# by ASCII characters
b = bytes([0, 46, 63, 133])
# raw and long literal are also supported
rb = b'\\x00abc\ndd'
rb = rb"""\x00abc
dd"""
zb = bytes(10) # 10 zero bytes´
```

rozhraní bytových polí se podobá rozhraní řetězců s těmito základními rozdíly:

`b[0]` → `0` [int]
indexace vrací položku (jako u seznamu) nikoliv jednobytové podpole

`b[0:1]` → `b'\x00'`: pro to lze využít výřez

`b.replace(b"\x00",b"!")` → `b'!.?\x85'`
všechny parametry musí být bytes

`b"-%d-%s" % (42, b"!")` → `` `b'-42-!' ``
formátování pomocí operátoru `%` (interpolace nepodporována)

bytes ↔ string

`b.decode("cp1250")` → `'\x00….?'`
`"ůňk".encode("latin2")` → `b'k\xf9\xf2'`
`"ůňk".encode("utf8")` → `b'k\xc5\xaf\xc5\x88'`
`"ůňk".encode("utf16")` → (16 bit s BOM)
`b'\xff\xfek\x00o\x01H\x01'`

proudy (stream, file-like) objekty

umožňují čtení respektive zápis posloupnosti bytů (*bytové proudy*) nebo znaků (*znakové proudy*) proudy z/do externích pamětí a komunikačních kanálů

metody proudů: `write`, `read`, `close`, `flush`
textové navíc: `readline`, `readlines`, `writeline`

proudy nad soubory v souborovém systému

uložení textu do souboru (kódování utf-8)

```
twriter = open("testfile", "wt")
# file name, mode=w(rite) t(ext)
twriter.write("žťčluouký\ůňnk\n")
twriter.flush() # clear buffer to file
twriter.close() # flush and close stream
```

čtení z textového souboru (kódování utf-8)

```
treader = open("testfile", "rt") #r(ead)
contents = treader.read() # #read all text
treader.close()
```

`contents` → `'žťčluouký\ůňnk\n'` [string], 14 zn.

alternativní čtení (po novém otevření):
`treader.read(1)` → `"Ž"`: čte jeden znak
`treader.readline()` → `'ťčluouký\n'\`

binární čtení (z textového souboru)

```
breader = open("testfile", "rb") #b(inary)
bcontents = breader.read() # read all bytes
breader.close()
```

`bcontents` pole bytů [bytes], 20 bytů
→ `b'\xc5\xbelu\xc5\xa5ou\xc4...\x88\n'`

alternativní čtení (po novém tevření):
`breader.read(1)[0]` → `197`: první byte jako `int`
`breader.read(24)` → `b'\xbelu\xc5...'`
požadováno 24 bytů, přečteno je 19 (víc jich není)

znaky jsou v UTF-8 kódovány jako jeden či dva byty.

Ž(17e)		l (6c)	u (75)	znaky
c5	be	6c	75	byty (hex)
11000101	10111110	01101100	01110101	bity

testování dosažení (aktuálního) konce

`read(n)` vrací řetězec/pole s méně než n znaky/byty

```
treader = open("testfile", "rt") # text m.
c = treader.read(1) # only one char
while c: # identical to len(b)!=0
  print(c)
  c = treader.read(1) # next char
```

3.8 lze zjednodušit přiřazovacím/(mrožím) operátorem.

```
treader = open("testfile", "rt") # text m.
while c := treader.read(1):
  print(c)
```

nehrozí nebezpečí přeplnění paměti, je-li soubor extrémně dlouhý, ale pomalé. řešení: **blokové čtení**:

```
bs = 1024 # size of block
treader = open("testfile", "rt") # text m.
while len(block := treader.read(bs)) == bs:
    print(block) # block with full size
if block: # process last (smaller) block
    print(block)
```

`readline` vrací prázdný řetězec (i prázdný řádek obsahuje na konci znak odřádkování)

```
treader = open("testfile", "rt") # text m.
while line := treader.readline():
  print(line) # insert the second newline!
```

hrozí nebezpečí přeplnění paměti, je-li nějaký řádek extrémně dlouhý

uzavírání souboru

proud by měl být po použití uzavřen (vyprázdnění bufferů, uvolnění prostředků)

volání `close` nemusí být provedeno (skok, výjimka), proto je preferována konstrukce `with`.

```
with open("binfile", "wb") as bf:
  bf.write(bytes(range(256)))
  # stream is closed at end of block
```

další proudy

StringIO = textový výstup/vstup do/z řetězce

```
with io.StringIO() as sio:
  sio.write("Αχιλλευς")
  sio.write("Πατροκλος")
  s = sio.getvalue()) # saved string
```

`s` → `'ΑχιλλευςΠατροκλος'`

BytesIO = bytový výstup/vstup do/z pole bytů

```
a = numpy.random.randint(32, 127, 10,
               dtype=numpy.uint8)
with io.BytesIO(bytes(a)) as bio:
  b = bio.read()
```

`b` ⇝ `'=C@yQ>|O(,'`

Gzip = komprimované čtení a zápis

přímý zápis do souboru (text pomocí `print`):

```
with gzip.open("text.gz", "wt") as gs:
    print("hello world!", file=gs)
```

proudy lze skládat: komprimovaný textový výstup do pole bytů (přes *BytesIO*).

```
with io.BytesIO() as bs: # outer stream
  with gzip.open(bs, "wt") as gs: # inner
    for _ in range(100):
      gs.write("Silor") # write 500 chars
  b = bs.getvalue() # get output as bytes
```

`len(b)` → `36`: 500 char/bytes to 36

manažer kontextů (context mng.)

objekty, které musí před použitím volat vstupní (*enter*) metodu a po něm výstupní (*exit*).

nejčastější použití: alokace prostředků (nepovinně v *enter*) a uvolnění (povinně v *exit*).

používají se v konstrukci `with`, viz výše proudy

```
with os.scandir(".") as entries: # entry
  for entry in entries: #inside context
      print(entry.name)
# object entries release resources  (exit)
```

objekt v *entries* je zároveň iterátor i manažer kontextu

výpočetní kontext (výpočet s nastavenou přesností):

```
from decimal import Decimal, localcontext
px = Decimal("1") / Decimal("42")
with localcontext() as ctx:
    # change only in block context
  ctx.prec = 10 # precision only 10 digits
  mpx = Decimal("1") / Decimal("42")
# exit to standard precison
```

`px` →
`Decimal('0.02380952380952380952380952381')`
`mpx` → `Decimal('0.02380952381')`

Python III — výraz a příkaz

výrazy

výraz je zápis, jehož vyhodnocením se získá hodnota. Vedlejším efektem může být změna objektů či proměnných, ale v Pythonu to není běžné.

výraz může obsahovat:
literály: přímé zápisy hodnot (čísel, řetězců)
proměnné: pojmenované odkazy na objekty
přístupy k podobjektům: atributy, indexy
volání funkcí a metod
uzávorkované výrazy: výrazy v kulatých závorkách, zápisy kolekcí (seznamy, slovníky, množiny,n-tice) resp. komprehenze (kulaté závorky lze často vynechat)
operátory: prefixové či infixové zápisy operací

pořadí vyhodnocení určuje následující tabulka:

`(...), [...], {...}`	uzávorkované výrazy
`x.y, x[index], f()`	přístup, volání funkce
`await x`	asynchronní výraz
`**`	mocnění
`+x,-x,~x`	unární prefix. operátory
`*,/,//,%`	multiplikativní operátory
`+,-`	aditivní operátory
`<<,>>`	bitové posuny
`&`	bitové and
`^`	bitové xor
`!`	bitové or
`<,>,==,in,is`	relační operátory
`not x`	logické not
`and`	logické and
`or`	logické or
`if-else`	podmínkový výraz
`lambda`	lambda výraz
`:=`	přiřazovací výraz

lambda výraz

vyhodnocením lambda výrazu vznikne anonymní (nepojmenovaná) funkce

`lambda` `seznam-parametrů` **`:`** `výraz`

výsledek λ-výrazu je objekt — lze jej ukládat, předávat do funkcí, či vracet.

```
s = ["Frodo", "Sam", "Gandalf"]
s.sort(key=lambda item: len(item))
['Sam', 'Frodo', 'Gandalf']
```

v Pythonu lze vždy místo λ-výrazu využít definici funkce (**`def`**), opačně to neplatí (tělem funkce je vždy výraz).

prázdný příkaz – pass

nutný v případě, že je očekáván odsazený blok, ale žádný sémantika nevyžaduje příkaz

```
class MyInt(int): #identical to base class
  pass

def donothing():
  pass # empty function
```

přiřazovací příkaz

umožňuje do (pojmenovaného) paměťového místa (proměnné, atributu, indexace) přiřadit odkaz na objekt

- `pam-místo = výraz`
- `n-tice-pam-míst = n-tice`
- `výřez = seznam`

některé binární operátory (mocnění, aditivní, multiplikativní, bitové) podporují tzv. **složené přiřazení**:

`pam-místo OP= výraz`

je ekvivalentní běžnému přiřazení

`pam-místo = pam-místo OP ( výraz )`

`i += 1` je ekvivaletní `i = i + 1`
`i *= a + b` je ekvivalentní `i = i * (a + b)`

volání funkce/metody

volání funkce/metody, jež má tzv. **postranní efekt**:
- I/O operace (např. **`print`**)
- změna stavu objektu (typicky adresáta metody)
- změna globální/nelokální proměnné (nepříliš košer)
- `id-funkce ( spec-parametrů? )`
- `pam-místo.id-metody ( spec-parametrů? )`

`s.sort()`: setřídí seznam `s` (změna adresáta metody)

pokud funkce/metoda vrací objekt je zahozen (výjimkou je použití v interakt. prostředí, kdy je vypsán)

příkaz `math.sin(5)` má smysl jen v interakt. prostředí (např. *Jupyter notebook*). Funkce `sin` nemá postranní efekt a vrací číslo (ve skriptu nemá žádný efekt).

příkaz assert

definuje tzv. **testovací tvrzení** (podmínku). Není-li splněno, je program ukončen výjimkou *AssertionError*, jež může zahrnovat řetězec zprávy (doporučeno).

`assert` `podmínka , řetězec-zprávy`

```
def length_interval(min, max):
  """length of interval [min, max]"""
  assert min <= max, "max is below min"
  return max - min
```

využití:
- **vstupní podmínky**: co musí splňovat parametry funkcí resp. počáteční stav objektu na počátku metody
- **výstupní podmínky**: co musí splňovat návratová hodnota nebo stav objektu na konci metody (lépe využít *jednotkové testování*).
- **invarianty**: stav, jenž musí průběžně platit (ani zde nejsou aserce otimálním řešením).

v „optimalizovaném“ módu jsou aserce ignorovány

```
python -O file.py # without assertion
```

větvení if

větvení na základě splnění podmínky (podmínek)

```
if podmínka :
        odsazený-blok
(elif podmínka :
        odsazený-blok)*
(else :
        odsazený-blok´)?
```

vykonán je odsazený blok (větev) za **první pravdivou** podmínkou (vždy jen jeden). Pokud není žádná pravdivá, je vykonán blok za **`else`** (je-li přítomen).

větvení match 3.10

větvení na základě hodnoty nebo struktury objektu.

```
match výraz :
        (case vzor :
                odsazený-blok)+
        (case _ :
                odsazený-blok)?
```

vykonán je odsazený blok (větev), jehož **vzor** (za `case`) jako první odpovídá **hodnotě** výrazu (za **`match`**). Pokud žádný neodpovídá, je vykonán blok se vzorem `_` (podtržítko), je-li přítomen.

nejjednodušší jsou vzory tvořené jednoduchými literály (čísla, řetězce). Ty odpovídají, pokud se rovnají hodnotě.

vzory lze řetězit pomocí svislítka (stačí když je plněn alespoň jeden podvzor).

```
match status:
  case 400: # if status is equal to 400
    return "Bad request"
  case 418: # if status is equal to 418
    return "I'm a teapot"
  case 401 | 403 | 404:
  # if status is 401 or 403 or 404
    return "Not allowed"
  case _: # otherwise
    return "Something's wrong with world"
```

vzory v podobě **literály n-tic/seznamů** mohou obsahovat:
- **literály**: musí se shodovat s odpovídající položkou hodnoty
- **proměnné:** odpovídají libovolnému objektu dané položky hodnoty, jež je následně navázán na proměnou
- **podržítka:** odpovídají libovolnému objektu dané položky hodnoty

vzor odpovídá hodnotě, má-li stejný tvar (počet položek na všech úrovních vnoření) a všechny položky si odpovídají (nezáleží na typu sekvence)

hodnota: `(1, ["a", "b"])`
vzory odpovídající hodnotě: `_`, `(_, _)`, `(1, _)`, `(1, s)` `s = ["a", "b"]`, `(_, ("a", _))`, `[x, ("a", y)] x = 1, y="b"`

podobně lze jako vzory užívat složené literály slovníků (musí si odpovídat položky se stejnými klíči)

```
match json_command:
   case {"action": "print", "value": v}:
      print(v)
   case {"action": "exit", "value": rv}:
      sys.exit(rv)
   case {"action": _, "value": v}:
      print("undefined action")
```

shodu (*match*) lze zpřesnit tzv. **strážemi** (*guard*) – doplňující podmínkou

```
match number_pair:
  case (x, y) if x > y:
    print("decreaing pair")
  case (x, y) if x < y:
    print("increasing pair")
  case (_, _):
    print("identical pair")
  case _:
    raise ValueError("invalid value")
```

cykly

cykly umožňují vícenásobně opakovat blok příkazů

for

```
for identifikátor in iterovatelný-objekt :
        odsazený-blok
```

cyklus **`for`** prochází přes všechny prvky iterátoru (získaného z iterovatelného objektu). Užívá se tehdy, je-li předem známa procházená posloupnost hodnot, či alespoň počet iterací.

iterace přes posloupnost čísel (výpočet $\sum_{i=1}^{9} \frac{1}{i}$

```
suma = 0
for i in range(1, 10): from 1 to 9 = 10-1
  suma += 1 / i
```

iterace přes obsah kolekce:

```
s = [1, 2, 3]
for item in s:
  print(s)
```

při iteraci přes kolekci nelze prvky měnit ani přesouvat

iterace přeskakující prvky lze dosáhnout pomocí výřezů, ale může být méně efektivní (např. u polí), efektivní je však zpětná iterace pomocí reverzního iterátoru.

```
s = [1, 2, 3]
for item in reversed(s):
    # better than s[::-1]
  print(s)
```

while

`while` `podmínka :`
`        odsazený-blok`

cyklus **`while`** se provádí, dokud je splněna podmínka (není-li splněna na začátku neprovede se ani jednou).

počet iterací nemusí být předem znám

```
while x > 0:
  print(x % 2)
  x //= 2
```

skoky

Python podporuje pouze tzv. strukturní skoky tj. zevnitř konstrukcí vně

opuštění cyklu/iterace

break: ukončuje (opouští) nejvnitřnější cyklus

continue: ukončuje aktuální iteraci nejvnitřnějšího cyklu

```
for i in range(8):
  if i == 5:
    break # exit from loop
    print(i)
# print 0, 1, 2, 3, 4
for i in range(8):
  if i == 5:
    continue # next iteration
    print(i)
# print 0, 1, 2, 3, 4, 6, 7
```

předčasné (nepredikovatelné) ukončení iterací:

```
with open("test.txt", "rt") as f:
  for line in f:
    if line.strip() == "":
      break # only to first empty line´
    print(line)
```

cyklus s podmínkou na konci:

```
while True:
  l = int(input("Write random number: "))
    if l == 42:
      break
print("OK")
```

return

ukončuje vykonávání těla funkce

`return` `výraz?`

nejčastěji s argumentem (návratová hodnota funkce):

```
def cross(a,b):
  return a*-b
```

výjimečně bez argumentů pro předčasný výskok:

```
def find_end():
  with open("test,txt", "rt") as f:
    for line in f:
      for word in line.split(" "):
        if word == "end":
          return # exit both loops
        print(word)
```

definice funkce

definice funkce vytváří funkci a váže ji na identifikátor (odsazený-bok = tělo funkce není provedeno)

`def` `identif ( specifikace parametrů ):`
`   odsazený-blok`
`specifikace-parametrů` `:=`
`    (striktně-poziční-parametry , /´,`)?
`    běžné-parametry?`
`    (*argv? |`
`    (, *,` `striktně-pojmenované-parametry)?)`
`    **kwargv?`

skutečný parametr: výraz uvedený ve volání funkce
formální parametr: proměnná lokální ve funkci, do níž se uloží **odkaz** na hodnotu skutečného parametru resp. odkaz na implicitní hodnotu

při volání lze **běžné form. parametry** nejdříve naplnit nepojmenovanými skut. parametry (podle pozice), zbývající pak pojmenovanými skut. parametry (uvedenými v libovolném pořadí).

```
def find(needle, haystack):
  return needle in haystack
```

- vše pozičně: `find("2", "42")`: hodnoty form. parametrů: `needle=2`, `haystack="42"`
- smíšeně: `find("2", haystack="42")`
- vše jménem: `find(needle="2", haystack="42")`

striktně poziční form. parametry 3.8 lze předávat jen pozičně, **striktně pojmenované** jen jménem

```
def neighbourhood(x, /, epsilon, *, side):
  if side == "both":
    return (x - epsilon, x + epsilon)
  elsif side == "left":
    return (x - epsilon, x)
  elsif side == "right:
    return (x, x + epsilon)
```

přípustné jsou jen dva zápisy skut. parametrů: `neighbourhood(0, 1e-9, side="both")` resp. `neighbourhood(0, epsilon=1e-9, side="both")`

proměnný počet parametrů

do `*argw` form. parametru se ukládají všechny přebytečné nepojmenované skut. parametry, do `**kwargv` všechny přebytečné skut. parametry předané jménem.

```
def p(x, y, *args, s, **kwargs):
  print(args)
  print(kwargs)
```

při volání `p(1, 2, 3, s=5, q=6)` je `args=(3,)` a `kwargs={"q": 6}`

skut. parametry z sekvence/slovníku

zápisem `*výraz` lze sekvenci předat jako jednotlivé pozviční skut. parametry

```
point = (1, 2) # also [1, 2]
math.atan2(point) ≡ math.atan2(1,2)
```

zápisem `**výraz` lze slovník předat jako jednotlivé pojmenované skut. parametry

```
config = dict(size=12, family="Arno")
setf(**config) ≡ setf(size=12, family="Arno")
```

implicitní hodnoty parametrů

implicitní hodnoty form. parametrů jsou použity, není-li ve volání uveden příslušný skut. parametr
- impl. hodnoty musí být nemodifikovatelné (či alespoň neměněné) [jsou vytvářeny je jednou při def. funkce]
- impl. hodnota si vynucuje použití impl. hodnot u všech následujících potenciálně pozičních form. parametrů

```
def trace(text, file=sys.stderr):
  print(text, file=file)
```

další příkazy

- **`import`** viz program a balíčky
- `with` viz II. manažer kontextů
- **`del`** viz I. list (operátory/příkazy)
- **`try`**, **`raise`** viz výjimky

Python IV — třídy

objekt je pamětovou representací/modelem reálné entity. Objekt:
• je jednoznačně identifikovatelný
• má interní stav
• na podněty reaguje změnou stavu a/nebo vytvořením nového objektu
• má omezenou životnost (typicky explicitně vzniká a automaticky zaniká)

třída representuje model množiny objektů se stejným chováním (reagujícími stejně na stejné podněty)

objekt je instancí třídy

definice třídy

v realitě je objekt primární a třída sekundární v programu je to naopak (praktický nominalismus) – nejdříve je nutno definovat třídu a pak je možné vytvářet objekty

```
class PaperSheet:
  # class attributes
  A4 = (210, 297)
  A5 = (148, 210)
  # constructor
  def __init__(self, width, height):
      assert width > 0, "Invalid width"
      assert height > 0, "Invalid height"
    self.w = width
    self.h = height
  # method
  def fold(self, orientation = "height"):
    if orientation == "height":
      self.h //= 2
    else:
      self.w //= 2
  # private (auxiliary method)
  def _format_spec(self):
    return f"{self.w}×{self.h}"
  # special (dunder) method
  def __str__(self): # string represent.
    return f"sheet: {self._format_spec()}"
  # property getter
  @property
  def area(self):
    return self.w * self.h
  # static method
  @staticmethod
  def fromSize(size):
    return PaperSheet(*size)
  # class method
  @classmethod
  def rectangular(cls, length):
    return cls(length, length)
```

constructor

speciální metoda `__init__`) inicializuje objekt = vytváří atributy objektu a přiřazuje do nich (pod)objekty.

`self`: první povinný parametr všech instančních metod, odkazuje na objekt nad nímž se metoda volá (jméno parametru je *ex more* vždy `self`)

```
p = PaperSheet(200,300)
```

atributy objektu

vnitřní proměnné objektu s odkazy na podobjekty.

přístup **uvnitř metod třídy**: `self.w` (čtení i zápis)

přístup **vně třídy** je omezen principem **zapouzdření** = atributy jsou neveřejné (ukrytí implementačních detailů + zajištění konzistence)

získání je v mnoha případech OK: `x = p.w` (zcela neveřejné atributy často začínají podtržítkem)

změna mimo třídu je podporována jen výjimečně. Python mu však nebrání `p.w = -1` (syntakticky správně, navzdory narušení konzistence)

(instanční) metody

mění stav objektu nebo vrací na základě jeho stavu nový objekt (výjimečně obojí)

volání uvnitř metod třídy: `self.fold()`

volání vně třídy: `p.fold()`

většina metod je součástí tzv. (veřejného) rozhraní. Některé metody (zde `_format_spec`) ale mohou být **neveřejné** (nejsou v dokumentaci, a běžně začínají podtržítkem).

speciálním typem instančních metod jsou **dunder metody** (*d[ouble] under[scores]*). Definují standardní rozhraní (protokoly) a jsou speciálně volány/aktivovány.

metoda `__str__` je volána konstruktorem řetězce: `str(p)` (volání `p.__str__()` je možné, ale neužívané)

vlastnosti (property)

metoda pro získání stavu objektu (tzv. (*getter*)), která se navenek tváří jako čtení/získání atributu

může být doplněna sesterskou metodou pro nastavení stavu (tzv. (*setter*)) se syntaxí přiřazení do atributu

vlastnosti jsou **syntaktické pozlátko**, lze je nahradit běžnými metodami (ale mnohdy jsou čitelnější)

v Pythonu lze read-only vlastnost získat vložením dekorátoru `@property` před tělo vhodné metody (nesmí mít kromě `self` žádný parametr a musí vracet nějakou vlastnost/stav objektu)

volání: `p.area` (bez dekorátoru by se volala jako běžná metoda tj. `p.area()`)

pomocí vlastností lze elegantně řešit omezení zapouzdření (ukrytí atributů a zajištění jejich konzistence)

řešení s dvojicí běžných metod (méně elegantní):

```
class PositiveInt:
  def __init__(self, val):
    self._v = set_value(val)
     # private attribute set by setter!
  def get_value(self):
      # public getter method
    return self._v
  def set_value(self, val):
      # public setter method
    assert val > 0, "Invalid value"
    self._v = val
pv = PositiveInt(1)
```

získání (čtení): `pv.get_value()`
změna (zápis): `pv.set_value(100)`

řešení pomocí vlastnosti s *getter* a *setter* metodou

```
class PositiveInt
  def __init__(self, val):
    self.value = val
     # property (not attrib.) is set
  @property
  def value(self): # getter
    return self._v
  @value.setter
  def value(self, val): # setter
    assert val > 0, "Invalid width"
    self._v = val # private attribute
pv = PositiveInt(1)
```

´ získání (čtení): `pv.value`
změna (zápis): `pv. = 100`
na rozdíl od přímé změny atributu vede nesplnění k assercí (= výjimce) `pv.value = -1`. Neprojde ani použití konstruktoru `PositiveInt(-1)` (využívá property setter).

třídní atributy

třídní atributy patří třídě nikoliv jednotlivým instancím. Při použití se uvnitř i vně třídy kvalifikují jménem třídy.

```
size = PaperSheet.A4
```

statické metody

statické metody jsou běžné funkce kvalifikované jménem třídy. Nevolají se nad instancemi a nemají povinný parametr `self`.

statické metody využívají dekorátor **`staticmethod`**.

často se využívají jako **tovární metody** pro tvorbu objektů: `PaperSheet.fromSize(PaperSheet.A4)`

další využití je manipulace s třídními objekty

```
class CountedInstance:
  counter = 0
  @staticmethod
  def count_new_object(): #
    CountedInstance.counter += 1
  def __init__(self):
    CountedInstance.count_new_object()

o1 = CountedInstance()
o2 = CountedInstance()
print(CountedInstance.count) # print 2
```

třídní metoda

třídní metody jsou volány nad třídou nikoliv instancemi tříd. Třída je předána jako první parametr (`cls`) namísto instance.

```
PaperSheet.rectangular(100)
```

použitím (a voláním) se podobají statickým metodám jsou však pružnější (viz dědičnost)

využívá dekorátor **`classmethod`**

metody nad třídami mohou podporovat pouze jazyky, v nichž je třída objektem, **třída(objekt) ≠ instance třídy** (třídou všech tříd je třída **`type`**)

přehled běžných „dunder“ metod

převod na řetězec

`__str__(self)`: převod na lidmi čitelný řetězec (pro ladění)
`str(datetime.date.today())` → *`'2022-02-11'`*

`__repr__(self)`: převed na strojově čitelný řetězec (typicky parsovatelný do původní podoby)
`repr(datetime.date.today())`
→ *`'datetime.date(2022, 2, 11)'`*

není-li `__str__` použije se místo něj `__repr__`, není-li definována ani jedna, je vypsáno jméno třídy.

`__format__(self, fmt)`: formátování (použité např. v interpolovaných řetězcích)

`format``(42, "3x")` = *`f"{42:3x}"`* → *`' 2a'`*

převod na elemantární typy

`__int__(self)`: přetypování na **`int`**
`int``("42")` → 42

`__float__(self)`: přetypování na **`float`**
`int``("42.0")` → 42.0

`__bool__(self)`: přetypování na **`bool`**
není-li, použije se **`len`**`(self) !=0`, jinak vždy `True`.
`bool``("")` → `False` (**`len`**`("")` je 0)

porovnání a uspořádání

hodnotové porovnání (protokol equateble) vyžaduje definovat metodu `__eq__(self, other)` (metoda `__ne__` vrací implicitně negaci `__eq__`). Nitné požadavky viz II. equatable. Jinak je použito odkazové porovnání.

pokud je definována metoda `__eq__` musí být definována i metoda `__hash__` pro výpočet hashovací hodnoty (použita v indexaci slovníků).

T: pokud `x == y` pak také **`hash`**`(x) == `**`hash`**`(y)`

Pokud mají instance třídy podporovat navíc uspořádání je třeba navíc implementovat `__gt__` (operátor >), `__lt__` (<), `__ge__` (>=) a `__le__` (<=).Nutné požadavky viz II. orderable.

náisledující třída poskytuje řetězce, které se porovnávají a uspořádavají jen podle prvního znaku. Vlastní provedení je delegováno na vnitřní objekt třídy **`str`**.

```
class FCStr:
  def __init__(self, text):
    self.t = text
  def __eq__(self, other):
    return self.t[0] == other.t[0]
  def __hash__(self):
    return hash(self.t[0])
  def __str__(self):
    return f"[{self.t[0]}]{self.t[1:]}"
  def __gt__(self, other):
    return self.t[0] > other.t[0]
  def __lt__(self, other):
    return self.t[0] < other.t[0]
  def __ge__(self, other):
    return self.t[0] >= other.t[0]
  def __le__(self, other):
    return self.t[0] <= other.t[0]
```

dědičnost (inheritance)

dědičnost nabízí prostředky jak vytvářet nové třídy specializací, rozšíření či skládáním již existujících. Navíc nabízí jednoduše dosažitelný **polymorfismus**.

jednoduchá dědičnost

jednoduchá dědičnost je binární relace:
bázová třída (*nadtřída*) poskytuje atributy a jejich inicializaci a také metody, která nad objekty odkazovanými metodami pracují
odvozená třída (*podtřída*) dědí atributy s jejich inicializací a může přidávat další. Může tak dědit i metody bázové třídy. Může přidávat další metody, ale také původní metody nově implementovat (= **předefinovávat**, (*override*)). Předefinované metody musí splňovat původní kontrakt (= mít požadované chování, i když mají jiný kód).

instance odvozené třídy musí být schopni zastoupit instance původní třídy tzv. *substituční princip Liskovové*

další podmínky:
• nové či předefinované metody musí využít, pokud možno, všech zděděných atributů
• neměly by být předef. všechny metody bázové třídy

jednoduchá dědičnost je uspořádání tříd tj. antisymetrická a tranzitivní relace (od odvozené třídy lze odvodit další třídu, která je pak nepřímo odvozená od původní základní)

zavádí **hierarchii mezi třídami**, která je v Pythonu **stromem**. Kořenem je třída **`object`**. Od ní jsou implicitně odvozeny třídy, pokud nestanoví jinak (v realitě je to spíš keř s několika delšími větvemi)

```
class SimpleParaWriter: # base class
    def __init__(self, filename):
        self.out = open(filename, "wt")
    def __enter__(self):
        return self
    def __exit__(self, e_t, e_v, e_tb):
        self.out.close()
    def writeline(self, text):
        self.out.write(text + "\n")

class HtmlParaWriter(SimpleParaWriter):
    def __init__(self, stream, parTag="p"):
        super().__init__(stream)
        self.tag = parTag
    def writeline(self, text, escape=True):
        if escape:
            text = html.escape(text)
        self.out.write(f"<{self.tag}>\n")
        super().writeline(text)
        self.out.write(f"</{self.tag}>")
    def writeempty(self):
        self.out.write(f"<{self.tag}/>\n")

with HtmlParaWriter("output", "div") as w:
    w.writeline(">>x<<")
    w.writeempty()
```

konstruktor musí explicitně volat konstruktor bázové třídy (funkce´ **`super`** vrací stejný odkaz jako `self` avšak s typem bázové třídy) – **delegování**

odvozená třída `HtmlParaWriter` dědí atribut `self.out`, včetně spec. metod `__enter__` a `__exit__` tj. je to *správce kontextu* (lze jej používat s `with`).

v nových/předefinovaných metodách je možno volat původní metody pomocí funkce **`super`**.

důsledky substutučního principu:
• předef. metody mohou mít další parametry, ty však musí mít implicit. hodn. nebo být strikt. pojmenované
• zděděné parametry předef. metod musí podporovat objekty stejné třídy nebo nadtřídy
• návratové hodnoty předef. hodnoty musí být stejné třídy nebo podtřídy
• musí platit vstupní asserce nebo jejich méně strikt. verze
• musí platit výstupní asserce nebo jejich striktnější verze

vícenásobná dědičnost

třída může bezprostředně dědit (metody) z více nadtříd

základní pravidlo: lze volat jakoukoliv metodu ze všech přímých nadtříd (a jejich nadtříd atd. tranzitivně), pokud více nadtříd definuje příslušnou metodu, pak o volané rozhoduje MRO (*Method Resolution Order*).

```
class A:
    def m(self):
        print("class A")
class B(A):
    def m(self):
        print("class B")
class C(A):
    pass # no new or overrided method
class D(B, C):
    def superm(self):
        super().m()
```

`D.mro()` → `[<class 'D'>,<class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>]`
`D().m()` vypíše **`class B`**

metoda **`super`** vrací hodnotu třídy, která je druhá v MRO (následující)
`D().supermm()` vypíše **`class B`**

problematické je delegování konstruktorů (mohou být volány vícenásobně) a tak se využívají specifické typy vícenás. dědičnosti např. **mixiny**.

Python V — třídy 2, výjimky

mixiny

mixiny jsou (jednoduché) třídy, které přidávají (mix-in) učitou funkčnost (běžné) třídě prostřednictvím mechanismu **vícenás. dědičnosti**.

nemají žádný konstruktor a tím ani vlastní atributy

mixiny by měli v MRO přecházet rozšiřované třídě, tj. píší na začátku seznamu bázových tříd

```
class ObjectInfoMixin:
  def obj_info(self):
    cname = self.__class__.__name__
    instid = id(self)
    strval = str(self)
    return f"{cname}({instid:x})={strval}"

class InfoInt (ObjectInfoMixin, int):
# mix-in 'obj_info' method into 'int' class
    pass # all is done, no new method
InfoInt(2).obj_info()
```
→ `InfoInt(7fdf3a926340)=2`

běhová identifikace tříd

vlastní třídu každého objektu lze získat pomocí atributu `__class__` (příklad výše)

užitečnější z hlediska **polymorfismu** je zjištění, zda je objekt instancí dané třídy či tříd odvozených (tranzitivně) tj. nabízí určité rozhraní

`isinstance(2, int)` → `True` (přímá instance)
`isinstance(2, numbers.Integral)` → `True`
zahrnuje všechny celočíselné třídy
`isinstance(2, numbers.Real)` → `True`
bázová třída všech tříd, jejichž instance podporují operace typické pro reálná čísla (tj. i `int`)

nadtřídy užívané k identifikaci jsou často tzv. **abstraktní bázové třídy** (ABC).

• tvoří kořen dílčí hierarchie mající společné rozhraní • nemusí mít implementovány všechny metody (neexistuje žádná společná implementace)
• nelze vytvářet přímé instance (jen instance odvoz. tříd)

`numbers.Integral()` vede k výjimce ***TypeError***: *Can't instantiate abstract class...*

abstraktní třídy nejsou nezbytné pro polymorfismus jen pro identifikaci instancí tříd s požadovaným rozhraním/protokolem´

poznámka: abstraktní třída se v Pythonu může stát nadtřídou i tzv. registrací/adopcí

```
from collections.abc import Iterable
# p-norma of vector or scalar (= abs)
def norm(v, p=2.0):
  if isinstance(v, numbers.Complex):
    return abs(v)
  elif isinstance(v, Iterable):
    return sum(x**p for x in v) ** (1/p)
  else:
    raise TypeError("Unsupported type")
```

větvení podle polymorfních typů podporuje i konstrukce `match` 3.10

```
def norm(v, p=2.0):
  match v:
    case numbers.Complex():
      return abs(v)
    case collections.abc.Iterable():
      return sum(x**p for x in v) ** (1/p)
    case _:
      raise TypeError("Unsupported type")
```

```
norm([1, 2, 3], 3) → 3.3019272488946263
norm(range(100)) → 573.0183243143276
norm(-1) → 1
```

polymorfismus zajišťuje, že funkce `norm` funguje např. i pro NumPy pole (neboť jsou iterovatelná):

```
numpy.arange(0,100) → 573.0183243143276
```

výsledek je ale funkční jen pro jednorozměrná pole a je pomalý. Lze však vložit i větev pro specializované podtřídy (pořadí větví je v tomto případě důležité!)

```
...
  elif isinstance(v, numpy.ndarray):
    return sum(v ** p) ** (1.0/p)
  elif isinstance(v, Iterable):
  # less effective but more general
    return sum(x**p for x in v) ** (1/p)
...
norm(numpy.random.rand(3,3))
```
⇝ (norma pro každý řádek matice 3×3)
`[1.29111178 1.28611962 1.33600064]`

specializované třídy

Python pomocí dědičnosti a dekorátorů podporuje i dvě specializované třídy, jež automaticky poskytují některé standardní *dunder* metody.

enumerations (výčty)

výčtové třídy (*enumerations*) mají **limitovaný počet instancí** opatřených **symbolickými jmény**.

```
class SColor(enum.Enum): # based on Enum
  GREEN = 0 # values are irrelevant
  YELLOW = 1 # but they have to be unique
  RED = 2
```

přístup k instanci (= třídnímu atributu): `SColor.GREEN`

klíčovou operací je **testování shody** (porovnání). Navíc jsou podporovány dunder metody `__str__`, `__repr__` (liší se).

`SColor.GREEN == SColor.RED` → `False`

efektivnější je referenční provnání:

`SColor.RED is SColor.RED` → `True`
`str(Color.GREEN)` → `'GREEN'`
`repr(Color.GREEN)` → `'<SColor.GREEN: 0>'`

třída je iterovatelná (pře všechny instance):

`list(SColor)` →
`[<SColor.GREEN: 0>, <SColor.YELLOW: 1>,...]`

instance jsou hashovatelné (lze je využít jako klíč ve slovníku)

`SColor.RED: "stop", SColor.YELLOW: "go"}`

instance jsou dostupné podle:
zástupné hodnoty:
`Color(1)` → `<SColor.YELLOW: 1>`
podle symbolického jména (řetězce):
`Color["RED"]` → `<SColor.RED: 1>`

pokud mají číselné hodnoty sémantický význam lze výčet odvodit z `enum.IntEnum`

```
class ExamResult(enum.IntEnum):
    Excelent = 1
    VeryGood = 2
    Good = 3
    Failed = 4
```

tento výčtový typ poskytuje navíc přetypování na [int] a uspořádání (podle číselných hodnot)

`int(ExamResult.Excelent)` → `1`
`ExamResult.Excelent < ExamResult.Good`
→ `True`

posledním specifickým typem výčtu je `IntFlag`. Ten umožňuje representovat i hodnoty vytvářené bitovými operacemi `&`, `|`, a `~` (a také `^` (tj. lze vytvářet objekty s více symbolickými hodnotami).

```
class Perm(enum.IntFlag):
  R = 4
  W = 2
  X = 1
  RW = R | W # optional composed values
```

`Perm.R | Perm.X` → `<Perm.R|X: 5>`
`(Perm.R | Perm.X) & ~Perm.X | Perm.W` →
`<Perm.RW: 6>`
`oct(Perm.RW)` → `'0o6'`

dataclasses (datové třídy)

Datové tříd usnadňují tvorbu a použití instancí jejichž jedinou funkcí je spojit více (pojmenovaných) hodnot do jednoho objektu (tzv. přepravka či box)

jedná se o flexibilnější alternativu k n-ticím

```
@dataclasses.dataclass(frozen=True)
class Point:
    x: float = 0.0 # instance attribute
    y: float = 0.0
```

pří definice se uvádí jen jména instančních atributů, jejich typ a případně implicitní hodnoty. Typ atributů je sice povinný ale je to jen (typová) anotace (typ se za překladu ani běhu nekontroluje).

dekorátor vytvoří konstruktor a metody pro porovnání a uspořádání a také převod na řetězec `__repr__`.

`str(Point(2, 3))` →
`'Point(x=2, y=3)'` (podobjekty jsou [int]!)
`repr(Point(y=3.0))` → (*x* je implicitně 0.0)
`Point(x=0.0, y=3.0)'`

porovnání a uspořádání je obdobné n-tici (uspořádání je lexikografické).

Parametr `frozen` s hodnotou `True` v dekorátoru vytváří nemodifikovatelné instance (doporučené, implicitní hodnota `frozen` je ale `False`)´

implicitní hodnoty nesmí být měnitelné objekty (jsou vytvářeny jen jen jednou a sdíleny všemi instancemi). Řešení (default factory je funkce volaná pro inicializaci atributu v každé instanci):

```
@dataclass
class ListCover:
    lst: list[int] = field(
              default_factory=list)
    # not list: list[int] = []
```

výjimky

výjimky nabízejí elegantní mechanismus pro řešení výjimečných situací

výjimečná situace: stav programu, jenž nelze vyřešit v kontextu (např. funkci/metodě), v níž vznikl (typicky chybový stav)

řešení výjimečné situace má dvě základní fáze:
1. detekce výjimečného stavu, vytvoření objektu výjimky a její vyhození
2. zachycení výjimky a využití informací v ní obsažených pro oštření (zachycení je běžně v jiném a vzdáleném kontextu).

čtyři základní ošetření:
1. ukončení programu (+ výpis dodatečných informací)
2. spuštění alternativního (náhradního) kódu (jež nevyužívá kód, v němž nastala výj. situace)
3. nové volání kódu, v němž vznikla výjimečná situace (musí se vytvořit nový kontext, nelze se vrátit do opuštěného kontextu)
4. vyhození nové (obecnější) výjimky (dočasné řešení)

vytvoření a vyhození výjimky

vyhození výjimky je de facto **vzdání se odpovědnosti** (něco se divného se děje a já nevím, co s tím). Po vyvolání se až do předání řízení obslužné rutině neprovádí běžný program.

`raise` třída-výjimky `(` param-konstruktoru `)`

třídou výjimky musí být třída `BaseException` nebo třídy z ní (i nepřímo) odvozené

jediným univerzálně využívaným parametrem kontextu je zpráva (určená primárně pro vývojáře a ladění)

standardní knihovna definuje několik univerzálněji použitelnějších tříd výjimek:
`Exception`: základní (nesystémová) výjimka
`LookUpError`: nenalezení prvku v kolekci (chybný index, klíč viz podtřídy `IndexError`, `Key´Error`)
`NotImplementadError`: metoda nemůže být či (ještě) není implementována
`RuntimeError`: obecná běhová výjimka (typicky nedostatek prostředků)
`TypeError`: neočekávaný typ (třída), tj. volání operace (ani v budoucnu) nepodporované danou třídou
`ValueError`: neočekávaná hodnota (správného typu)

pokus je výjimka vyhozena v reakci na zachycení jiné výjimky je možné původní výjimku přivěsit k nové (*exception chain*):

`raise` nová-výjimka `from` původní-výjimka

nebojte se vyhazovat výjimky, výjimka je vždy lepší než nedefinovaný stav programu!

zachycení a ošetření výjimky

výjimku lze zachytit v konstrukci `try` a následně ošetřit v jedné z navazujících sekcí `except`

využití výjimek a try-bloků umožňuje využít přístup: nejdříve to zkusím, a pokud to špatně dopadně pak to nějak (ex post) vyřeším. Je to ve většině případů vhodný přístup, ale může výrazně zpomalit běh programu.

```
try:
        odsazený-blok
(except třída-výjim. (as identif.)? :
        odsazený-blok)+
(except :
        odsazený-blok)?
```

výjimky vzniklé za běhu v try-bloku (včetně z bloku volaných metod, tranzitivně) jsou zachyceny a ošetřeny první sekcí `except` označené třídou výjimky resp. její nadtřídou (tranzitivně) (vždy je provedena nejvýše jedna sekce). Pokud neodpovídá žádná ze sekcí, je výjimka předána nadřízenému try-bloku.

pokud `except` sekce obsahuje část `as ident.`, je výjimka přiřazena do příslušného identifikátoru (a lze ji tak využít v obslužné rutině).

poslední `except` nemusí obsahovat jméno třídy, ošetřuje všechny (zbývající) výjimky (tj. jako by byla uvedena třída `SystemException`).

není-li výjimka zachycena v žádném z try-bloků pak ji zachytí obslužný kód a ukončí program s výpisem informace o výjimce na standardní chybový vstup.

```
def exceptional_func():
    raise RuntimeError("I am a mamoth!")

try:
    print("this will be displayed")
    exceptional_func()
    print("and this will not")
except RuntimeError as e:
# first more specific class
    print(f"We have a runtime problem.")
    print(f"Detail: {e}")
    sys.exit(1) # exit of program
except Exception as e:
# more general exceptions
    print(f"We have a unspecified problem.")
    print(f"Detail: {e}")
    sys.exit(1) # exit of program
```

finally

try-blok může následovat sekce `finally`.

kód sekce `finally` je vykonán:
• při jakémkoliv opuštění try-bloku (dosažením konce, výskokem, vyhozením [ne]nezachycené výjimky)
• vždy jako poslední (po kódu bloku či případně provedené obslužné rutině)

kód `finally` běžně nezachycuje výjimky

kód sekce `finally` by neměl obsahovat výskok (příkazy `return`, `break`, `continue`) – neintuitivní chování

`finally` se primárně využívá pro uvolnění prostředků alokovaných v try-bloku (ty se při předčasném ukončení bloku automaticky neuvolňují)

u většiny tříd s touto sémantikou se ale nyní využívá konstrukce `with` (ta interně využívá `finally`).

```
def func(i):
    try:
        if i == 0:
            return
        elif i == 1:
            raise ValueError()
        elif i == 2:
            raise TypeError()
        elif i == 3:
            raise Exception()
        print("end of try-block")
    except ValueError:
        print("catched exception")
    except TypeError as e:
        print("new chained exception")
        raise Exception() from e
    finally:
        print("always printed")
```

sekce `finally` je vykonána vždy, při předčasném výskoku (i==0), vzniku a zachycení výjimky (´1), vzniku výjimky při obsluze jiné výjimky (2), vzniku nezachycené výjimky (3) a také při dosažení konce bloku (i>3) a to vždy jako poslední ve funkci.

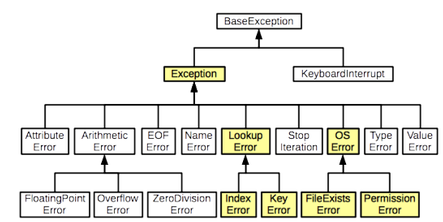
vlastní třída výjimek

vlastní třídy výjimek lze vytvářet odvozením ze třídy `Exception` a jejich podtříd (tranzitivně)

vhodné je implementovat konstruktor přijímající nepovinný parametr s textem zprávy.

```
class EmotionalException(RuntimeError):
    def __init__(self, msg = "bad emotion"):
        super().__init__(msg)

raise EmotionalException("I am stuck")
```



Python VI — proměnné

literál = přímý zápis objektu, identické literály vytvářejí shodný objekt (u měnitelných objektů) u neměnných mohou vracet odkaz na jediný (= identický) objekt.

proměnná = textový identifikátor odkazující na objekt (proměnná ≠ objekt). Proměnná může postupně odkazovat na různé objekty a objekt může být odkazován více proměnnými (i v jediném okamžiku).

objekty jsou dostupné jen tehdy jsou-li odkazovány prostřednictvím proměnných, atributů odkazovaných objektů nebo položek odkazovaných kolekcí. V opačném případě mohou být uvolněny z paměti.

```
x = 42
y = 42
```

`id(x)` ⇝ `93955157926656`
jednoznačná identifikace objektu (adresa v paměti)
`id(y)` ⇝ `93955157926656`
obě proměnné odkazují na stejný objekt

```
a = [1, 2]
b = [1, 2]
```

`id(a)` ⇝ `139820786929344`
`id(b)` ⇝ `139820786930496`
rozdílné objekty (prozatím) shodné
`a == b` → `True`, `a is b` → `False`

modul

pythonský modul je tvořen zdrojovým souborem s příponou `.py`. Název modulu je shodný se jménem souboru bez přípony.

Modul lze vykonat dvěma způsoby:
1. přímým spuštěním prostřednictvím interpreteru
2. importováním prostřednictvím příkazu **`import`** (jakou součást provádění jiného modulu)

výsledkem vykonávání je:
1. změna prostředků OS (IO zařízení, grafické objekty na displeji, apod.)
2. vznik a následný zánik objektů v paměti
3. vytváření **jmenných prostorů**, mapujících identifikátory na objekty (včetně funkcí/metod, tříd

většinu modulů má dvě části:
definiční = definice proměnných, funkcí, tříd: vykonávána při importu i přímém spuštění
výkonná = interakce objektů a komunikace s okolím: vykonávána jen při přímém spuštění.

```
import math

# definition part
def f(lx): # definition of function
  return lx*lx

x = 42 # definition of variable

if __name__ == "__main__":
  # executive part
  # only in direct launch
    sx = math.sqrt(x)
  print(f(sx))
```

proměnná `__name__` obsahuje při přímém spuštění řetězec `"__main__"` při importování název modulu

jmenné prostory

jmenný prostor je zobrazení identifikátorů (symbolů) na objekty (jež jsou tak pomocí identifikátorů identifikovány).

identifikátor je do jmenného prostoru přidán definicí, tj: • příkazem **`def`** (definice funkcí) • přiřazením do proměnné (není-li už součástí jmenného prostoru) • navázáním proměnné v konstrukcích **`for`**, **`with`** a **`except`** a v rámci aplikace vzoru v **`match`** • explicitní definicí identifikátoru v příkazech **`global`** a `blocal` • příkazem **`class`** (definice tříd)

dalším způsobem rozšíření jmenného prostoru je importování příkazem **`import`**

jmenné prostory jsou organizovány do stromu, v němž má každý jmenný prostor (kromě kořenového) nadřízený jmenný prostor

identifikátor, který není v aktuálním jmenném prostoru, se hledá v nadřízeném (tranz.).

globální jmenný prostor

identifikátory definované mimo těl funkcí a tříd patří do tzv. **globálního jmenného prostoru modulu**

do tohoto jmenného prostoru dále patří:
• importované identifikátory
• několik pomocných (dunder) proměnných spjatých s modulem např. `__name__`

seznam identifikátorů aktuálního jmenného prostoru:
`dir()` →

```
['__builtins__', '__file__',  '__name__',
...,  'f', 'math', 'sx', 'x']
```

jmenný prostor v podobě (read only) slovníku:
`globals()` →

```
{... math': <module 'math' (built-in)>,
'f': <function f at 0x7fd21ffa8280>,
'x': 42, 'sx': 6.48074069840786}
```

nadřízeným jmenným prostorem globálního jmenného prostoru je **jmenný prostor vestavěných (*builtins*) funkcí** (včetně std. tříd výjimek).

`dir(__builtins__)` → `['ArithmeticError', 'AssertionError', ..., abs, all, ..., zip]`

lokální jmenný prostor

lokální jmenný prostor se vytváří při každém provádění těla funkce/metody. Běžně zaniká po dosažení konce těla.

Kromě identifikátorů definovaných v těle funkce zahrnuje i formální parametry funkce.

```
def geomean(a,b): # formal parameter
    import math # imported module
    c = math.sqrt(a*b) # variable defin.
    return c
```

`dir()` → (na konci těla funkce geomean)
`['a', 'b', 'c', 'math']`
`locals()` → `{'a': 2, 'b': 3, 'math': <module 'math' (built-in)>, 'c': 2.449489742783178}`
v těle funkce vrací lok. jm. prostor

(bezprostředně) nadřízeným jmenným prostorem lokálního jmen. prostoru je jmenný prostor, v němž je umístěna definice funkce. Typicky je to globální jmenný prostor modulu a ve funkci tak lze přistupovat ke globálním identifikátorům a vestavěným funkcím

zápis do proměnné však vytváří (novou) lokální proměnou, i když existuje stejnojmenná globální (ta je pak tzv. **zastíněna** tj. není přístupná z lok. jmen. prostoru)

zápis do globálních proměnných v těle funkcí umožňuje příkaz **`global`**, jenž explicitně zpřístupňuje globální identifikátory pro zápis.

`global` `identifikátor+`

změna globálních proměnných jako vedlejší efekt funkcí není doporučená. Pokud funkce (a jejich volání) potřebují sdílet stav, lze využít třídu (metody sdílejí atributy objektu).

```
gx = 0 # definitions in global namespace
gy = 0
gz = 0

def function():
    global gx
    gx = 1 # global gx is changed
    gy = 1 # new local gy
    print(gx) # print global (1)
    print(gy) # print local  (1)
    print(gz) # print global (0)


function()
print(gx) # print 1
print(gy) # print 0
```

uzávěr (closure)

pokud je **funkce definována uvnitř těla jiné funkce** (= vnější funkce), pak je lokální jmen. prostor vzniklý (každým jednotlivým) voláním vnější funkce nadřízeným jmen. prostorem ve (všech) voláních funkce vnořené.

je-li vnořená funkce vrácena jako návratová hodnota vnější funkce stává se její nadřízený jmenný prostor **uzávěrem** ((*closure*)).
• lokální jmenný prostor, přeživší ukončení volání (vnější) funkce, jež ho vytvořila
• je sdílen všemi voláními vnořené funkce
• je viditelný pouze v těchto voláních

```
def outerf(x):
    def nestedf(y):
        # y is in local namespace
        # x is in outer namespace
        return x+y
    return nestedf


inc = outerf(1) # new func. with closure
dec = outerf(-1) # new func. with closure
```

`inc(5)` → 6, `dec(5)` → 4

Python umožňuje měnit hodnoty proměnných vnějších lok. jmen. prostorů (i uzávěrů). Příslušné proměnné musí být definovány příkazem `nonlocal` (obdoba **`global`**).

```
def make_sumator():
  suma = 0
  def sumator(item): # update closure
    nonlocal suma # required before assign
    suma += item # suma in outer namespace
  def result(): # get suma from closure
    return suma
  return sumator, result
  # pair of func. with shared closure
sm, rs = make_sumator()
sm(5) # add 5 to suma in closure
sm(4)
```

`rs()` → 9 = hodnota proměnné ve sdíleném uzávěru

uzávěry nabízejí prostředek pro bezpečné sdílení a ukrytí dat. Python je však využívá jen řídce (preferován je sdílení prostř. OOP objektů).

oblast viditelnosti (scope)

oblast viditelnosti identifikátorů tvoří kód (= řádky programu), v němž lze k identifikátoru přistupovat. V Pythonu je jednoznačně určen mechanismem jmenných prostorů:

kód je v oblasti viditelnosti identifikátoru ⇔ je při jeho provádění za běhu programu identifikátor dostupný v aktuálním jmen. prostoru resp. prostorech nadřízených.

globální identifikátor (identifikátor, jež se stane součástí glob. jmenného prostoru) je viditelný od místa své definice/importu až do konce modulu s výjimkou míst, kde je zastíněn (tj. kde je zaveden stejnojmenný lokální) identifikátor

lokální identifikátor je viditelný od místa své definice/importu až do konce těla příslušné funkce s výjimkou míst, kde je zastíněn.

• parametry funkce jsou viditelné až v těle funkce (tj. implicitní hodnotou parametru nemůže být jiný parametr)
• proměnná definovaná přiřazením nemůže být použita na pravé tohoto přiřazení(⇒ proměnnou nelze definovat složeným přiřazením)

Python **nepodporuje oblasti viditelnosti užší než funkce**. Řídící proměnná cyklu **`for`**, proměnné definované v konstrukci `with` jsou viditelné i mimo tyto konstrukce (i když jsou de facto nepoužitelné). Podobně lze i za konstrukcí **`match`** přistupovat k navázaným symbolům úspěšného vzoru.

importování modulů

příkazy **`import`**:
1) vykonají importovaný modul (hlavní cíl: vytvoření glob. jmenného prostoru)
2) rozšíří jmenný prostor importujícho modulu

import objektu modulu

`import` `jméno-modulu`

importující jmenný prostor je rozšířen o identifikátor importovaného modulu, jenž odkazuje na objekt modulu, jehož atributy jsou převzaty ze jmenného prostoru importovaného modulu

```
import math
```

funkce a proměnné definované v modulu jsou dostupné zápisem např. `math.sqrt` resp. `math.pi`.

výhoda: importující jmenný prostor je rozšířen o jediný identifikátor

import objektu modulu s aliasem

`import` `jméno-modulu` **`import`** `alias`

obdoba výše uvedeného, objekt modulu je však dostupný jen přes alias (ten jediný rozšíří importující jmenný prostor)

```
import math as m
```

přístup: `m.sqrt`, `m.pi` (nelze psát `math.pi` apod.)

u některých modulů se využívá pravidlem a aliasy jsou konvenční:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

import jednotlivých identifikátorů

`from` `jméno-modulu` **`import`** `seznam-identif.`

do importujícího jmenného prostoru jsou přidány všechny identifikátory jmenného prostoru importovaného modulu uvedené v seznamu

seznam lze uzavřít do kulatých závorek (může tak být víceřádkový)

```
from math import sqrt, pi
```

přístup: `sqrt` resp. `pi` (nikoliv `math.sqrt`)

ne**výhoda:** větší množství importovaných identifikátorů (z různých modulů) může vést ke kolizi identifikátorů

namísto seznamu-identifikátorů lze uvést znak `*`, a importovat tak všechny identifikátory nezačínající podtržítkem (nelze doporučit).

umístění importu

standardní umístění importů je na začátku modulu, alternativně je však lze umístit:

a) v obsluze výjimky `ImportError` jako fallback

```
try:
  from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree

print(type(etree.fromstring("<x></x>")))
```

pokud není k dispozici modul `lxml` je importován modul `xml.etree.ElementTree` pod stejným aliasem. Výsledkem parsování XML tak mohou být objekty dvou různých tříd (s obdobným rozhraním).

b) na začátku těla funkce/metody

modul je v tomto případě importován do lokálního jmenného prostoru a to jen v případě, že je funkce volána

výhoda: není-li funkce volána, modul nemusí být přítomen (instalován)

nevýhoda: import (= vykonání imp. modulu) se provádí při každém volání

```
def load_from_url(url):
    import requests # local import
    txt = requests.get(url).text
    return txt
```

korutiny

korutiny lze na rozdíl od rutin (běžných funkcí) dočasně opustit a přesunout se do volajícího kódu a poté se do korutiny vrátit (při zachování lok. jmenného prostoru)

rutina:
vstup: parametry (předány při volání) výstup: návratová hodnota
korutina:
vstup: při volání parametry a následně lze předat parametry při každém návratu do korutiny (nepříliš často využívané)
výstup: při volání se vrátí objekt tzv. generátoru, který nabízí rozhraní iterovatelného objektu a zároveň iterátoru, ten poté řídí přístup ke korutině a vrací postupně hodnoty předávané při dočasném opuštění korutiny (argumenty `yield`). Návratová hodnota (pokud korutina skončí) je nevýznamná (běžně je to `None`)

implementace korutin v Pythonu je zaměřena na jejich použití pro tvorbu lenivých iterátorů:

```
def iterate(f, x, n):
    for _ in range(n):
        yield init # coroutine is suspended
        initval = f(x)
```

iterátor vrací posloupnost hodnot $x, f(x), f(f(x)), \ldots$.