Ještě k funkcím

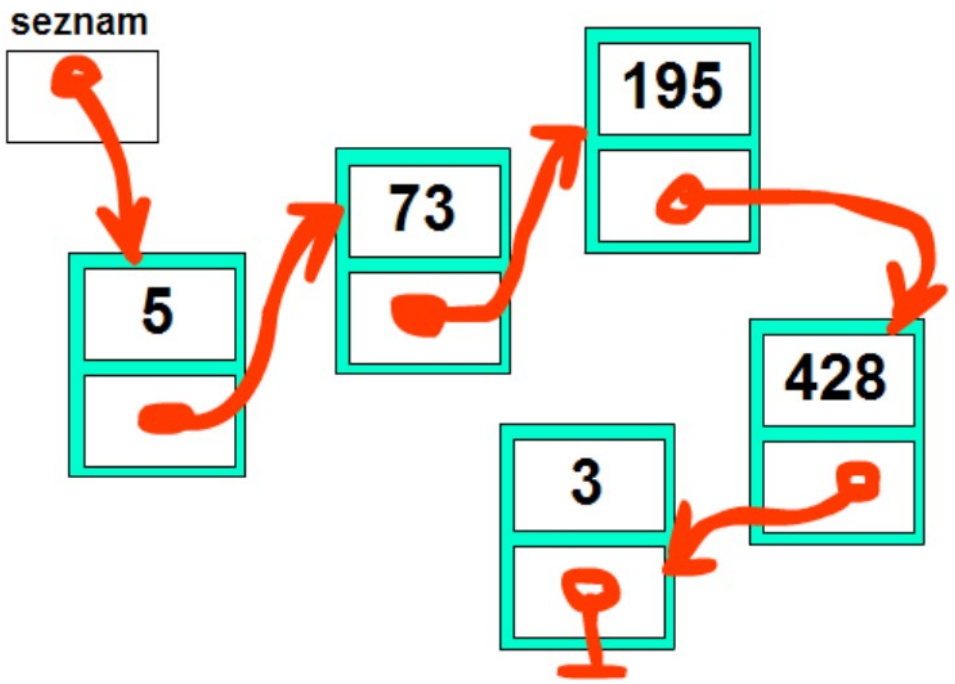
- Přístup ke globálním proměnným (a funkcím a...)
  - jenom pro čtení
  - pro čtení i pro zápis
- vnořené funkce: Přístup ke jménům z nadřazené funkce

Dynamické proměnné

- Proměnné, které vznikají (a zanikají) během výpočtu
- Mohou mít různou velikost
- Zvláštní oblast paměti = halda
- Nepotřebné proměnné se automaticky uklízejí
  (ne ve všech jazykách, ale v Pythonu ANO)
  = garbage collector
- Proměnná, která vypadá, že obsahuje třeba seznam,
  ve skutečnosti obsahuje pouze odkaz (adresu do haldy)
- Příklad: list, string, (dlouhé) celé číslo, slovník, objekt
- …ale také spojový seznam (vytvořený z objektů)
  nebo strom (taky vytvořený z objektů)
- prázdný ukazatel None (domluvená hodnota nikam neukazuju)

Lineární spojový seznam

- složený z prvků
- každý prvek obsahuje hodnotu a odkaz na další prvek



Lineární spojový seznam

```
class Prvek:
    def __init__(self, x, dalsi):
        self.x = x
        self.dalsi = další
```

```
p1 = Prvek(10, None)
p2 = Prvek(20, None)
p3 = Prvek(30, None)
p1.dalsi = p2
p2.dalsi = p3
```

...nebo:

```
p3 = Prvek(30, None)
p2 = Prvek(20, p3)
p1 = Prvek(10, p2)
```

...nebo taky:

```
seznam = Prvek(10, Prvek(20, Prvek(30, None )))
```

Lineární spojový seznam

```
def vytiskni(s):
    while s is not None:
        print(s.x, end=' ')
        s = s.dalsi

Vytiskni(seznam)
```

```
1 2 3
```

Lineární spojový seznam

...by taky mohl být objekt!

Operace...

- přidat prvek na začátek
- přidat prvek na konec
- vymazat první prvek
- vymazat poslední prvek
- ...

...a jejich složitosti.

Objekty a třídy (nejen v Pythonu) znovu a podrobněji

Objekty a třídy (nejen v Pythonu)

Pojmy:

- Třída je datový typ (formička)
- Objekt je instance (bábovička)

TH je objekt třídy Člověk

Jak je to v jazyku Python

Definice třídy

```
class Komplex:
    pass  # dál už nic není

k = Komplex()
k.Re = 3.00
k.Im = 4.00
print(k.Re, k.Im)
```

```
3.0 4.0
```

Přidat funkci

- má povinný první parametr `self` = TENHLE objekt, který (ten parametr) se při volání neuvádí

```
class Komplex:
    def abs(self):
        return (self.Re**2 + self.Im**2) ** (1/2)


k = Komplex()
k.Re = 3.00
k.Im = 4.00
print(k.Re, k.Im, k.abs()) # bez parametru!
```

```
3.0 4.0 5.0
```

Konstruktor

- funkce pro inicializaci nového objektu
- volá se při vytváření objektu
- vždycky se jmenuje `__init__`
- …takže může být jenom jeden
- má také povinný první parametr `self`
  (může se jmenovat i jinak, `self` je jenom zvyklost)
- když konstruktor neuvedeme,
  použije se standardní (prázdný) konstruktor

Konstruktor

```
class Komplex:
    def abs(self):
        return (self.Re**2 + self.Im**2) ** (½)

    def __init__(self, Re, Im):
        self.Re = Re
        self.Im = Im


k = Komplex(3.00, 4.00)
print(k.Re, k.Im, k.abs())
```

```
3.0 4.0 5.0
```

Dědičnost (inheritance)

- Při definici nové třídy můžeme říci,
  že je odvozená („zděděná“) od jiné třídy.
- Odvozená třída zdědí:
  - všechny datové součásti
  - všechny funkce
- Navíc můžeme definovat nová data i nové funkce
- Zděděné funkce můžeme změnit (předefinovat)

Dědičnost

```
class A:
    x = 10
    def tiskA(self):
        print("tiskA()")

class B( A ):
    def tiskB(self):
        print("tiskB() x=", self.x)
        self.tiskA()

b = B()
b.tiskA()
print()
b.tiskB()
```

```
tiskA()

tiskB() x= 10
tiskA()
```

Dědičnost – volání funkce předka

```
class A:
    def f(self):
        print("A.f()")

class B(A):
    def f(self):
        print("B.f()")
        super().f()

class C(B):
    def f(self):
        print("C.f()")
        super().f()

c = C()
c.f()
```

```
C.f()
B.f()
A.f()
```

Může objekt patřit do více tříd?

Hierarchické vztahy mezi třídami (ISA-vztah = "is a"):

pes je savec
savec je obratlovec (takže pes je taky obratlovec)
obratlovec je živočich (takže pes je taky živočich)

Ale nejenom to:

pes je savec
pes je domácí zvíře (a ne každé domácí zvíře je savec)
pes je přítel člověka (nejlepší)

TH je Učitel, Programátor, Muzikant, Vynálezce, Výtvarník…

Násobná dědičnost

- v různých programovacích jazycích
  lze deklarovat, že třída je pod-třídou jiné třídy
  („dědičnost"), potom dědí všechny prvky svého předka

- v některých programovacích jazycích (i v Pythonu)
  lze deklarovat, že třída je pod-třídou více tříd
  („násobná dědičnost"), potom dědí všechny prvky všech svých předků.

  Je s tím spousta potíží !! („diamond problem")

- proto v některých programovacích jazycích (např. *C#*)
  násobná dědičnost není a nahrazuje se pomocí interface

Interface („rozhraní“)

Na rozdíl od dědičnosti se nic nedědí,
jenom slibujeme, že objekt tyhle funkce bude mít/umět
(„objekt splňuje interface XYZ“)

Abstraktní datový typ

V Pythonu – viz Duck typing (dále).

Virtuální metody

V odvozené třídě lze předefinovat metodu tak,
že všechna její volání (i ze starších tříd a metod)
budou volat tu novou.

V Pythonu jsou všechny metody virtuální.

Podrobnosti – u nějakého jiného jazyka.

Polymorfismus

```
class Zvire:
    def __init__(self, jmeno):
        self.jmeno = jmeno
    def VydejZvuk(self):
        print("!@#$%^")

class Pes(Zvire):
    def VydejZvuk(self): # předefinujeme zděděnou funkci
        print("HAF:", self.jmeno)

class Kocka(Zvire):
    def VydejZvuk(self): # předefinujeme zděděnou funkci
        print("Mnau:", self.jmeno)

class Had(Zvire):
    def VydejZvuk(self): # předefinujeme zděděnou funkci
        print("Sssss:", self.jmeno)

class Kapr(Zvire):
    def VydejZvuk(self): # předefinujeme zděděnou funkci
        print("...:", self.jmeno)

zoo = [Pes("Archie"), Kocka("Babeta"), Had("Python"), Kapr("Karel")]
for z in zoo:
    z.VydejZvuk() # kdyby se někdo ptal, jestli z má funkci VydejZvuk(), tak má
```

```
HAF: Archie
Mnau: Babeta
Sssss: Python
...: Karel
```

Všechna zvířata mají konstruktor __init__(…),
zdědila ho od předka Zvire.

Abstraktní třída

- Třída, ze které nechceme vytvářet instance
- Slouží jenom jako společný předek
- Python sám neumí, řeší modul abc (Abstract Base Classes)
- V jazycích s typovou kontrolou je potřebná ke kontrole, jestli předáváme parametr správného/odpovídajícího typu
- V Pythonu se nekontroluje !@#$%^ (dynamické typování)…

Duck typing:

Když to chodí jako kachna, plave jako kachna
a kváká jako kachna… – tak je to kachna!

Kontrola typu

1. funkce type(…)

```
if type(th) != Herec: # Herec je jméno třídy
    print("vetřelec!!!")
```

2. funkce isinstance(…)

```
if not isinstance(p, Zvire):
    print("p není Zvíře!")
```

isinstance(…) vrací hodnotu True i pro odvozené typy

…ale v Pythonu se používá duck-typing.

*) typové anotace - možná někdy jindy

Zapouzdření (encapsulation)

Možnost určovat,
které součásti objektu budou viditelné zvenku (veřejné)
a které budou viditelné jenom uvnitř (soukromé).

Slouží pro zachování konsistence dat.

„Nedovolte nikomu sahat na svá data!"

…v Pythonu není !

- Náhražka: Když jméno začíná _ (podtržítko),
  tak byste to (u cizího objektu) neměli používat/volat
  (a neimportuje se příkazem import)
- Když jméno začíná __ (dvě podtržítka, „dunder"),
  a na konci má max. jedno podtržítko
  tak se __jméno přejmenuje na _JménoTřídy__jméno
  (tím se zabrání konfliktům jmen s rodičovskou třídou)

```
class Trida:
    __TajnaPromenna = 27


t = Trida()
print(t._Trida__TajnaPromenna)
```

Komu patří funkce...

Funkce definovaná v třídě může být

1) funkce instance/objektu
2) statická funkce
3) funkce třídy

1) Funkce patřící objektu

- patří objektu (volá se prostřednictvím objektu)
- má přístup k jeho funkcím a proměnným

= to, co jsme viděli doposud

2) Statická metoda/funkce

- patří třídě a ne objektu
  (volá se pomocí třídy i objektu)
- nemá přístup k proměnným objektu

(žádný objekt ani nemusí existovat)

- nemá přístup ani k proměnným třídy

```
class Trida:

    @staticmethod
    def sedm():
        return 7

print(Trida.sedm())
t = Trida()
print(t.sedm())
```

```
7
7
```

2) Statická metoda/funkce[2]

```
class K2:
    def E():
        return 6
    @staticmethod
    def F():
        return 7
    def G():
        return 8
```

```
>>> K2.E()
6
>>> K2.F()
7
>>> K2.G()
8
```

```
>>> k = K2()
>>> k.E()
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    k.E()
TypeError: K2.E() takes 0 positional arguments but 1 was given
>>> k.F()
7
>>> k.G()
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    k.G()
TypeError: K2.G() takes 0 positional arguments but 1 was given
```

3) Metoda třídy (class method)

- patří třídě a ne objektu (volá se prostřednictvím třídy)
- nemá přístup k proměnným objektu
  (žádný object ani nemusí existovat)
- má přístup k proměnným třídy

```
class AA:
    pocet = 0
    @classmethod
    def MetodaTridy(cls): # parametr cls
        cls.pocet += 1
        print(cls.pocet)

AA.MetodaTridy()
AA.MetodaTridy()
```

```
1
2
```

...nahradit více konstruktorů: static

(POZOR: Nevytváří objekt třídy POLE,
ale objekt třídy list.)

```
class POLE:
    @staticmethod
    def ZeStringu(s):
        return s.split()
    @staticmethod
    def Opakuj(hodnota, pocet):
        return pocet*[hodnota]

print(POLE.ZeStringu("a b c d e f g hijkl"))
print(POLE.Opakuj(25, 10))
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'hijkl']
[25, 25, 25, 25, 25, 25, 25, 25, 25, 25]
```

...nahradit více konstruktorů: class

```
class Komplex:
    def __init__(self, Re, Im):
        self.Re = Re
        self.Im = Im
    def Tisk(self):
        print("Re:", self.Re, "Im:", self.Im )

    @classmethod
    def Re(cls, Re):
        return cls(Re, 0)
    @classmethod
    def Im(cls, Im):
        return cls(0, Im)

Komplex(2,3).Tisk()
Komplex.Re(5).Tisk()
Komplex.Im(7).Tisk()
```

```
Re: 2 Im: 3
Re: 5 Im: 0
Re: 0 Im: 7
```

Proměnná definovaná v třídě může být

1) proměnná patřící instanci
2) proměnná patřící třídě

1) proměnná patřící instanci

Zatím všechno, co jsme viděli,
i když ve skutečnosti…

2) proměnná patřící třídě

```
class KLAS:
    pocet = 0
    def __init__(self):
        KLAS.pocet += 1
    def f(self):
        #self.pocet += 1
        return self.pocet

t = KLAS()
print(t.f(), t.f(), t.f())
s = KLAS()
print(s.f(), s.f(), s.f())
```

a) `#self.pocet += 1:`

```
1 1 1
2 2 2
```

b) `self.pocet += 1:`

```
2 3 4
3 4 5
```

Pokud v tom nemáte jasno, tak to nepoužívejte!

2) proměnná patřící třídě – doplnění

https://docs.python.org/3/reference/simple_stmts.html#assignment

Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, **the right-hand side expression, `a.x` can access either an instance attribute or** (if no instance attribute exists) **a class attribute**.
The **left-hand side target `a.x` is always set as an instance attribute, creating it if necessary.** Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the right-hand side expression refers to a class attribute, the left-hand side creates a new instance attribute as the target of the assignment:

```
class Cls:
    x = 3               # class variable
inst = Cls()
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
```

Proč objekty

Použití objektů nám dovoluje

nepřepisovat existující zdrojový kód

ale místo toho

odvodit novou třídu,
kde změníme (jen) to, co potřebujeme změnit.

Příklad s želvou…

dědičnost (IS A) x náležení/agregace (HAS A)