Základy (objektového) Pythonu

A4B99RPH – Řešení problémů a hry

Tomáš Svoboda, svobodat@fel.cvut.cz

katedra kybernetiky, centrum strojového vnímání

26. září 2012

Paradigmata programování, malá odbočka

Strukturované programování:

- poddruh imperativního programování
- posloupnost příkazů, určuje přesný postup
- podobný klasickým návodům, blízký lidskému uvažování

Objektově orientované programování:

- interakce objektů např. posíláním zpráv
- manipulace s daty (objekty)

Deklarativní programování: (funkcionální, logické, . . .)

- definujeme problém
- postup řešení najde jazyk/počítač sám

. . .

Paradigmata programování, malá odbočka

Strukturované programování:

- poddruh imperativního programování
- posloupnost příkazů, určuje přesný postup
- podobný klasickým návodům, blízký lidskému uvažování

Objektově orientované programování:

- interakce objektů např. posíláním zpráv
- manipulace s daty (objekty)

Deklarativní programování: (funkcionální, logické, . . .)

- definujeme problém
- postup řešení najde jazyk/počítač sám

. . .

Paradigmata programování, malá odbočka

Strukturované programování:

- poddruh imperativního programování
- posloupnost příkazů, určuje přesný postup
- podobný klasickým návodům, blízký lidskému uvažování

Objektově orientované programování:

- interakce objektů např. posíláním zpráv
- manipulace s daty (objekty)

Deklarativní programování: (funkcionální, logické, . . .)

- definujeme problém
- postup řešení najde jazyk/počítač sám

. . .

Funkce (procedury)

- *strukturování* programu
- logické celky
- opakující se operace (filipika proti copy+paste technice)

```
1 def jmeno_funkce(parametry):
2     prikazy
3     return(promenne)
```

Funkce (procedury)

- *strukturování* programu
- logické celky
- opakující se operace (filipika proti copy+paste technice)

```
1 def jmeno_funkce(parametry):
2     prikazy
3     return(promenne)
```

Funkce – příklad

```
1 def area(radius):
2     temp = 3.14159 * radius**2
3     return temp
```

Pochopitelně, že:

```
1 class Circle:
2     def __init__(self,radius):
3         self.r = radius
4
5     def area(self):
6         return(self.r**2 * 3.14159)
```

Funkce – příklad

```
1 def area(radius):
2     temp = 3.14159 * radius**2
3     return temp
```

Pochopitelně, že:

```
1 class Circle:
2     def __init__(self,radius):
3         self.r = radius
4
5     def area(self):
6         return(self.r**2 * 3.14159)
```

Funkce – v modulech

Předpokládejme funkce v souboru `knihovna.py`.

```
1 import knihovna
2 vysledek = knihovna.moje_funkce(parametry)
```

nebo

```
1 from knihovna import *
2 vysledek = moje_funkce(parametry)
```

případně

```
1 from knihovna import moje_funkce
2 vysledek = moje_funkce(parametry)
```

Funkce – v modulech

Předpokládejme funkce v souboru `knihovna.py`.

```
1 import knihovna
2 vysledek = knihovna.moje_funkce(parametry)
```

nebo

```
1 from knihovna import *
2 vysledek = moje_funkce(parametry)
```

případně

```
1 from knihovna import moje_funkce
2 vysledek = moje_funkce(parametry)
```

Funkce – v modulech

Předpokládejme funkce v souboru `knihovna.py`.

```
1 import knihovna
2 vysledek = knihovna.moje_funkce(parametry)
```

nebo

```
1 from knihovna import *
2 vysledek = moje_funkce(parametry)
```

případně

```
1 from knihovna import moje_funkce
2 vysledek = moje_funkce(parametry)
```

Funkce – vstupní parametry

```
1 def moje_fce(par1, par2=implicitni_hodnota_pro_par2):
2     prikazy
3     return(promenna)
4
5
6 vysledek_pro_default_par2 = moje_fce(par1_hodnota)
7 vysledek = moje_fce(par1_hodnota, par2_hodnota)
```

Jsou možné i další variance na dané téma (viz např. část 4.7 na http://docs.python.org)

Konvence, zvyklosti – krátké ohlédnutí

- 4 mezery pro odsazování (editory to typicky udělají za vás), nemíchejte tabelátory a mezery
- raději řádky kratší než 79 znaků
- nepoužívejte diakritiku v názvech souborů, jménech proměnných a raději ani v komentářích.
- jména funkcí a proměnných: `mala_pismena_s_podtrzitky`
- mezery:

```
1 hypot2 = x*x + y*y    NE    hypot2=x * x + y * y
```

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[1]

Konvence, zvyklosti – krátké ohlédnutí

- 4 mezery pro odsazování (editory to typicky udělají za vás), nemíchejte tabelátory a mezery
- raději řádky kratší než 79 znaků
- nepoužívejte diakritiku v názvech souborů, jménech proměnných a raději ani v komentářích.
- jména funkcí a proměnných: `mala_pismena_s_podtrzitky`
- mezery:

```
hypot2 = x*x + y*y    NE    hypot2=x * x + y * y
```

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[1]

Konvence, zvyklosti – krátké ohlédnutí

- 4 mezery pro odsazování (editory to typicky udělají za vás), nemíchejte tabelátory a mezery
- raději řádky kratší než 79 znaků
- nepoužívejte diakritiku v názvech souborů, jménech proměnných a raději ani v komentářích.
- jména funkcí a proměnných: `mala_pismena_s_podtrzitky`
- mezery:

```
1 hypot2 = x*x + y*y   NE   hypot2=x * x + y * y
```

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[1]

Konvence, zvyklosti – krátké ohlédnutí

- 4 mezery pro odsazování (editory to typicky udělají za vás), nemíchejte tabelátory a mezery
- raději řádky kratší než 79 znaků
- nepoužívejte diakritiku v názvech souborů, jménech proměnných a raději ani v komentářích.
- jména funkcí a proměnných: `mala_pismena_s_podtrzitky`
- mezery:

```
1 hypot2 = x*x + y*y    NE    hypot2=x * x + y * y
```

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[1]

Konvence, zvyklosti – krátké ohlédnutí

- 4 mezery pro odsazování (editory to typicky udělají za vás), nemíchejte tabelátory a mezery
- raději řádky kratší než 79 znaků
- nepoužívejte diakritiku v názvech souborů, jménech proměnných a raději ani v komentářích.
- jména funkcí a proměnných: `mala_pismena_s_podtrzitky`
- mezery:

```
1 hypot2 = x*x + y*y   NE   hypot2=x * x + y * y
```

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[1]

Konvence, zvyklosti – krátké ohlédnutí

- 4 mezery pro odsazování (editory to typicky udělají za vás), nemíchejte tabelátory a mezery
- raději řádky kratší než 79 znaků
- nepoužívejte diakritiku v názvech souborů, jménech proměnných a raději ani v komentářích.
- jména funkcí a proměnných: `mala_pismena_s_podtrzitky`
- mezery:

```
1 hypot2 = x*x + y*y    NE    hypot2=x * x + y * y
```

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[1]

Seznamy

Uspořádaná (multi)množina hodnot s indexem (od 0).

Příklady seznamů:

```
1 [10, 20, 30, 40, 40]
2 ["spam", "bungee", "swallow"]
3 ["hello", 2.0, 5, [10, 20]]
4 []
```

viz příklady on-line

Seznamy, procházení, příslušnost, ...

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i, horseman in enumerate(horsemen):
4         print(horseman, "is at position",i)
```

Pokud nepotřebujeme nutně index, pak elegantněji:

```
1 for horseman in horsemen:
2     print(horseman)
```

Seznamy, procházení, příslušnost, ...

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i, horseman in enumerate(horsemen):
4         print(horseman, "is at position",i)
```

Pokud nepotřebujeme nutně index, pak elegantněji:

```
1 for horseman in horsemen:
2     print(horseman)
```

Matice, aneb vnořené seznamy

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
1 matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2
3
4 >>> matrix[1]
5 [4, 5, 6]
6
7 >>> matrix[1][1]
8 5
```

Co vlastně označují proměnné?

```
1 a = [1,2,3]
2 b = a
3 b[1] = 10
4 print(b)
5 print(a)
```

N-tice

skoro jako seznamy, ale prvky N-tic jsou neměnné!

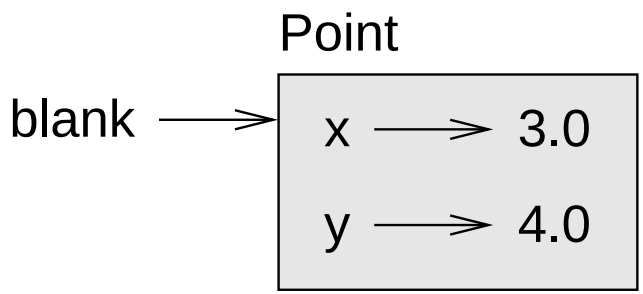
```
1 >>> ntice = (1,2,3)
2 >>> seznam = [1,2,3]
```

Neměnnost prvků může být i výhodou.

Třída jako rozšíření datových typů

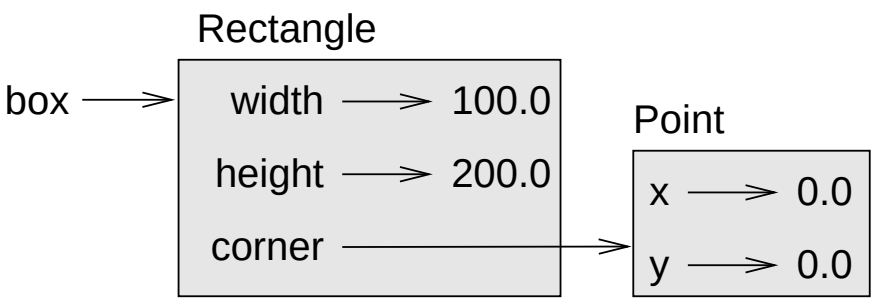
Python je velmi flexibilní (to může být výhoda i nevýhoda). A toto je nejjednodušší použití.

```
1 class Point:
2     """represents a point in 2-D space"""
3
4 blank = Point()
5 blank.x = 3.0
6 blank.y = 4.0
```



Vnořené (embedded) objekty

```
 1 class Rectangle:
 2     """represent a rectangle.
 3        attributes: width, height, corner.
 4     """
 5
 6 box = Rectangle()
 7 box.width = 100.0
 8 box.height = 200.0
 9 box.corner = Point()
10 box.corner.x = 0.0
11 box.corner.y = 0.0
```



Mělká a hluboká kopie složeného objektu

```
 1 class Point:
 2     """represents a point in 2-D space"""
 3
 4 class Rectangle:
 5     """represent a rectangle. attributes:
 6     width, height, corner"""
 7
 8 box1 = Rectangle()
 9 box1.width = 100
10 box1.height = 100
11 box1.corner = Point()
12 box1.corner.x = 1
13 box1.corner.y = 1
14
15 box2 = box1
16 box2.height = 200
```

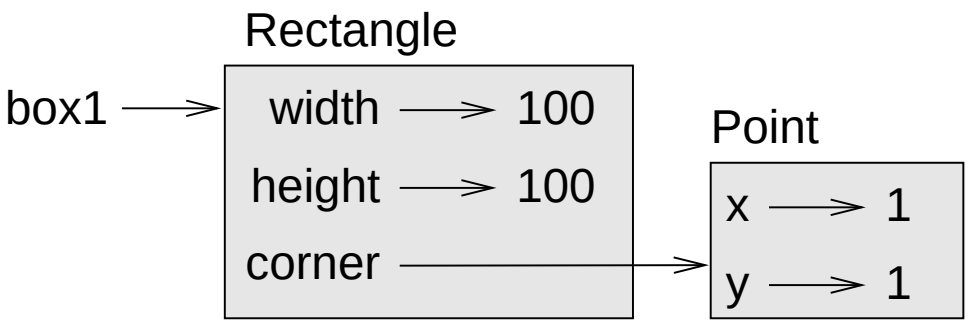
Mělká a hluboká kopie složeného objektu

Možný zdroj těžko odhalitelných chyb.

Zkusme kód a přemýšlejme, proč vidíme to, co při zběžném *mělkém* pohledu nedává smysl.

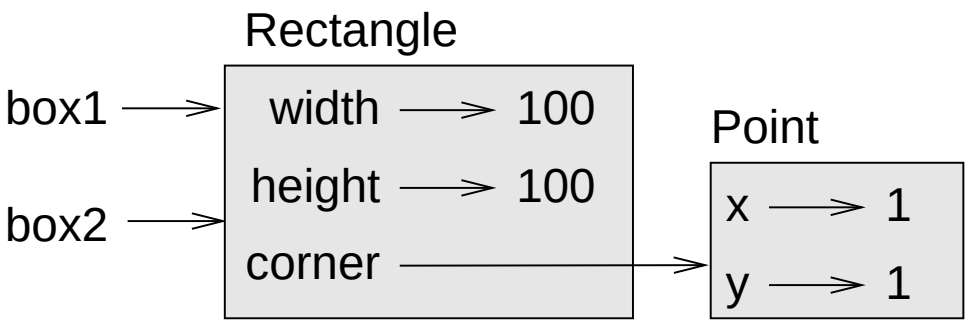
Přiřazení

```
1 box2 = box1
```



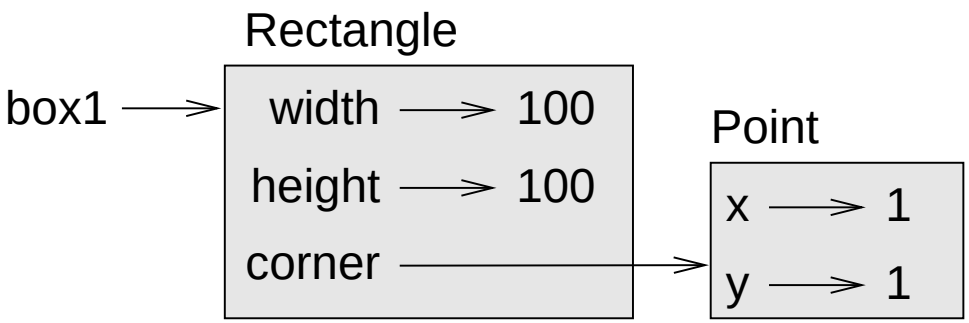
Přiřazení

```
1 box2 = box1
```



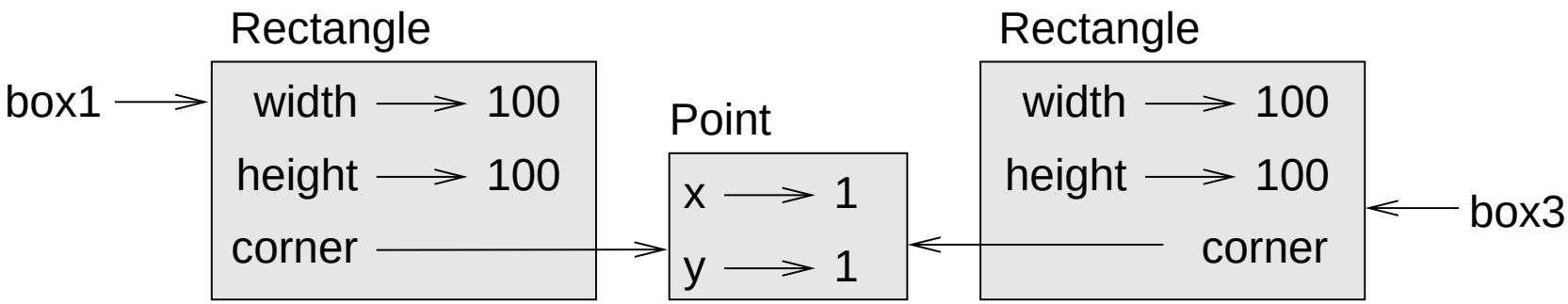
Mělká kopie

```
1 box3 = copy.copy(box1)
```



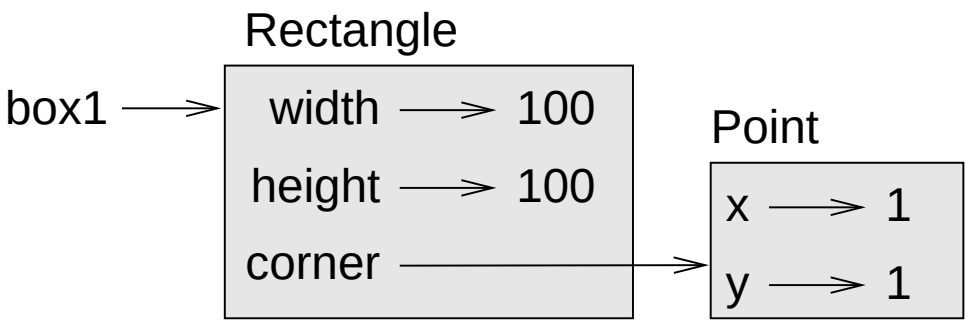
Mělká kopie

```
1 box3 = copy.copy(box1)
```



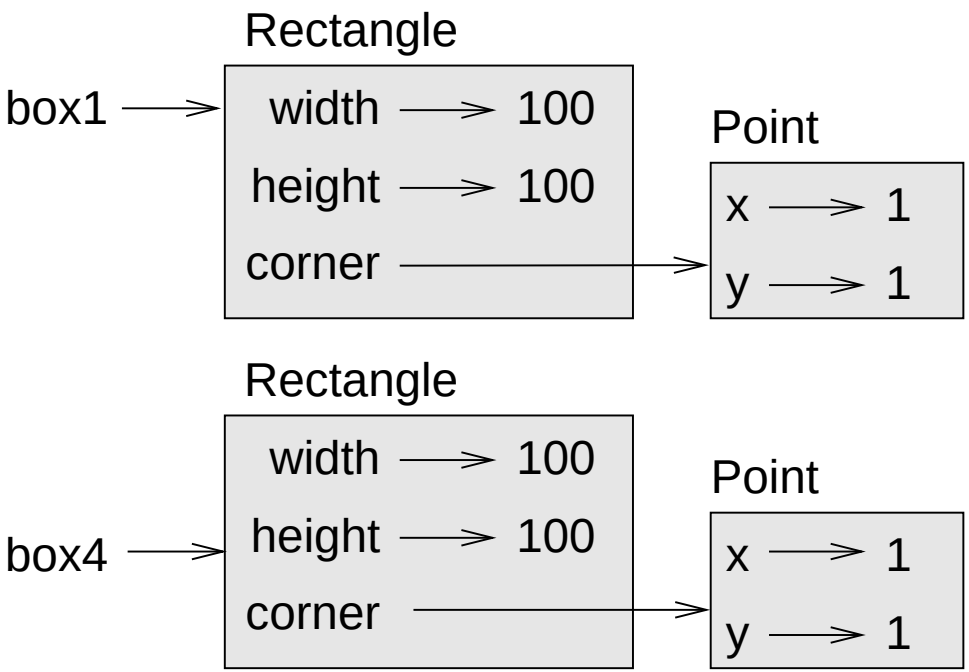
Hluboká kopie, aneb klonování

```
1 box4 = copy.deepcopy(box1)
```



Hluboká kopie, aneb klonování

```
1 box4 = copy.deepcopy(box1)
```



Třída Time

```
1 class Time:
2     """represents the time of day.
3        attributes: hour, minute, second"""
4
5 time = Time()
6 time.hours = 11
7 time.minutes = 59
8 time.seconds = 30
```

Produktivní (čistá) funkce

```
1 def add_time(t1, t2):
2     sum = Time()
3     sum.hours = t1.hours + t2.hours
4     sum.minutes = t1.minutes + t2.minutes
5     sum.seconds = t1.seconds + t2.seconds
6     return sum
```

- funkce nic nemodifikuje
- vytváří nový objekt
- a není dobře

Funkce – modifikátor

```
 1 def increment(time, seconds):
 2     time.seconds += seconds
 3
 4     if time.seconds >= 60:
 5         time.seconds -= 60
 6         time.minutes += 1
 7
 8     if time.minutes >= 60:
 9         time.minutes -= 60
10         time.hours += 1
```

Čistou funkci nebo modifikátor?

- čisté funkce generují obvykle čitelnější kód
- méně náchylné k chybám (pamatujeme diskusi okolo hloubky kopií)
- modifikátory mohou být někdy efektivnější
- ...

print_time

```
 1  class Time:
 2      """represents the time of day.
 3         attributes: hour, minute, second"""
 4
 5  def print_time(time):
 6      print("{:02d}:{:02d}:{:02d}".format(time.hours, time.
 7
 8  >>> start = Time()
 9  >>> start.hours = 9
10  >>> start.minutes = 45
11  >>> start.seconds = 00
12  >>> print_time(start)
13  09:45:00
```

poznámka k formátování print

Python 2.x vs 3.x

Funguje v Python 2.x i 3.x, ale označeno jako obsolete

```
1 print('%.2d:%.2d:%.2d' % (self.hours, self.minutes, self.
```

Nově:

```
1 print("{:02d}:{:02d}:{:02d}".format(self.hours, self.minu
```

viz: http://docs.python.org/py3k/library/string.html#string-formatting

též http://docs.python.org/py3k/library/stdtypes.html#old-string-formatting-operations

print_time jako metoda třídy

```
1 class Time:
2     def print_time(self):
3         print("{:02d}:{:02d}:{:02d}".format(self.hours, s
```

- v zásadě se změnilo jen odsazení
- *self* označuje objekt, kterého se to týká
- z příkladu použití to bude jasné

Čase, vytiskni se! vs. vytiskni nějaký čas

funkce jako aktivní prvek: „vytiskni nějaký čas (který je ti dán)“

```
1 start_time = Time()
2 print_time(start_time)
```

objekt jako aktivní prvek: „vytiskni se, zobraz se!“

```
1 start_time = Time()
2 start_time.print_time()
```

Čase, vytiskni se! vs. vytiskni nějaký čas

funkce jako aktivní prvek: „vytiskni nějaký čas (který je ti dán)“

```
1 start_time = Time()
2 print_time(start_time)
```

objekt jako aktivní prvek: „vytiskni se, zobraz se!“

```
1 start_time = Time()
2 start_time.print_time()
```

Čase vytiskni se – lépe!

```
1 class Time:
2     def __str__(self):
3         return("{:02d}:{:02d}:{:02d}".format(self.hours,
```

```
1 start_time = Time()
2 print(start_time)
```

Přetížení metody print

Čase vytiskni se – lépe!

```
1 class Time:
2     def __str__(self):
3         return("{:02d}:{:02d}:{:02d}".format(self.hours,
```

```
1 start_time = Time()
2 print(start_time)
```

Přetížení metody print

Čase vytiskni se – lépe!

```
1 class Time:
2     def __str__(self):
3         return("{:02d}:{:02d}:{:02d}".format(self.hours,
```

```
1 start_time = Time()
2 print(start_time)
```

Přetížení metody print

Modifikátor objektu

```
1  class Time:
2      def increment(self,seconds):
3          self.seconds = self.seconds + seconds
4          while self.seconds >= 60:
5              self.seconds = self.seconds - 60
6              self.minutes = self.minutes + 1
7          while self.minutes >= 60:
8              self.minutes = self.minutes - 60
9              self.hours = self.hours + 1
```

```
1  c_time = Time()
2  c_time.hours = 10
3  c_time.minutes = 42
4  c_time.seconds = 24
5  c_time.increment(500)
```

Modifikátor objektu

```
1  class Time:
2      def increment(self,seconds):
3          self.seconds = self.seconds + seconds
4          while self.seconds >= 60:
5              self.seconds = self.seconds - 60
6              self.minutes = self.minutes + 1
7          while self.minutes >= 60:
8              self.minutes = self.minutes - 60
9              self.hours = self.hours + 1
```

```
1  c_time = Time()
2  c_time.hours = 10
3  c_time.minutes = 42
4  c_time.seconds = 24
5  c_time.increment(500)
```

Atributy (proměnné) třídy

```
1 c_time = Time()
2 c_time.hours = 10
3 c_time.minutes = 42
4 c_time.seconds = 24
5 c_time.increment(500)
```

```
1 class Time:
2     def increment(self,seconds):
3         self.seconds = self.seconds + seconds
4         while self.seconds >= 60:
5             self.seconds = self.seconds - 60
6             self.minutes = self.minutes + 1
7         while self.minutes >= 60:
8             self.minutes = self.minutes - 60
9             self.hours = self.hours + 1
```

__init__

```
1 class Time:
2     def __init__(self, hours=0, minutes=0, seconds=0):
3         self.hours = hours
4         self.minutes = minutes
5         self.seconds = seconds
```

```
1 >>> time = Time()
2 >>> print(time)
3 00:00:00
```

```
1 >>> time = Time (9)
2 >>> print(time)
3 09:00:00
```

```
1 >>> time = Time(9, 45)
2 >>> print(time)
3 09:45:00
```

__init__

```
1  class Time:
2      def __init__(self, hours=0, minutes=0, seconds=0):
3          self.hours = hours
4          self.minutes = minutes
5          self.seconds = seconds
```

```
1  >>> time = Time()
2  >>> print(time)
3  00:00:00
```

```
1  >>> time = Time(9)
2  >>> print(time)
3  09:00:00
```

```
1  >>> time = Time(9, 45)
2  >>> print(time)
3  09:45:00
```

__init__

```
1 class Time:
2     def __init__(self, hours=0, minutes=0, seconds=0):
3         self.hours = hours
4         self.minutes = minutes
5         self.seconds = seconds
```

```
1 >>> time = Time()
2 >>> print(time)
3 00:00:00
```

```
1 >>> time = Time (9)
2 >>> print(time)
3 09:00:00
```

```
1 >>> time = Time(9, 45)
2 >>> print(time)
3 09:45:00
```

__init__

```
1 class Time:
2     def __init__(self, hours=0, minutes=0, seconds=0):
3         self.hours = hours
4         self.minutes = minutes
5         self.seconds = seconds
```

```
1 >>> time = Time()
2 >>> print(time)
3 00:00:00
```

```
1 >>> time = Time (9)
2 >>> print(time)
3 09:00:00
```

```
1 >>> time = Time(9, 45)
2 >>> print(time)
3 09:45:00
```

interní proměnné a metody

- pokud neřeknete jinak, je vše veřejné
- některé proměnné a metody jsou spíše pracovní, okolí o nich vedět nemusí nic
- pomocí __ můžete specikovat interní proměnné a metody

Příklad interní metody:

```
1 def __init_dimensions(self):
2     """computes some internal variables"""
```

Takovou funkci pak můžete volat pouze v těle třídy

```
1 self.__init_dimensions()
```

Dědění

```
1  class Solver:
2      def __init__(self,maze):
3          """abstract solver implements some methods common
4          self.path = []
5          self.maze = maze
6          self.final_reached = False
7
8  class DummySolver(Solver):
9      def __init__(self,maze):
10         """simplistic solver finds a path regardless the
11         Solver.__init__(self,maze)
```

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Konvence, zvyklosti – krátké ohlédnutí II

- jméno třídy začíná velkým písmenem, slova uvnitř také

```
1 class MySuperClass:
```

- pozor na odsazení, ale opět, editor vás hlídá
- i když konstruktor `__init__` není nutný, je nanejvýš doporučené ho používat.
- čisté (pure) funkce jsou lepší než modifikátory
- vede-li návrh na modifikátory, možná je nejvyšší čas, začít přemýšlet objektově
- pozor na hloubku kopií proměnných

Toto není kompletní seznam, přečtěte si Style Guide for Python Code[2]

Game class

Obtíže s objekty a s programováním vůbec

- Nevzdávejte to!
- zkusíme spolu projít nejčastější problémy
- self a co s ním?
- platnost proměnných
- self vs. „statická“ proměnná
- dynamicky vs. staticky typovaný programovací jazyk

Obtíže s objekty a s programováním vůbec

- **Nevzdávejte to!**
- zkusíme spolu projít nejčastější problémy
- self a co s ním?
- platnost proměnných
- self vs. „statická“ proměnná
- dynamicky vs. staticky typovaný programovací jazyk

Obtíže s objekty a s programováním vůbec

- **Nevzdávejte to!**
- zkusíme spolu projít nejčastější problémy
- **self** a co s ním?
- platnost proměnných
- self vs. „statická“ proměnná
- dynamicky vs. staticky typovaný programovací jazyk

Slovníky, typ dict

- kolekce dvojic klíč–hodnota
- podobné seznamům či nticím, kdy místo pořadového indexu máme obecný klíč
- vyskytuje se i v jiných jazycích, někdy se používá název hash array
- velmi užitečná datová struktura
- určitě si přečtěte kap. 12 na `http://howto.py.cz`
- spoustu užitečných metod, viz google: python dict methods

jednoduché použití slovníku

```
 1 import string
 2 import employee
 3 def read_csv_list(fname):
 4     employees = {}
 5     try:
 6         f_in = open(fname,'r')
 7     except:
 8         raise IOError("could not open %s for reading"%fna
 9     for line in f_in.readlines():
10         line = sanitize_string(line)
11         data = string.split(line,";")
12         felusername = data[4]
13         if (len(felusername)>1):
14             employees[felusername]=employee.Employee(data
15     return(employees)
```

Otázky?

- pokud si vzpomenete později, použijte diskusní fórum
- nebo čas na cvičení
- nebo si svoji otázku poznamenejte a položte ji na další přednášce

Otázky jsou velmi důležité.

Literatura I