Zpracování a vizualizace dat v prostředí Python

Úvod

Zdeněk VAŠÍČEK
Fakulta Informačních Technologií, Vysoké Učení Technické v Brně
Brno, Czech Republic
vasicek@fit.vutbr.cz

Cíl předmětu

- Obecným cílem je seznámit studenty s problematikou získávání, zpracování, analýzy a vizualizace dat v prostředí jazyka Python.
  - Představit, jak maximálně využít existující frameworky a techniky napříč celým spektrem procesu zpracování a analýzy dat.
- Student by měl získat obecný přehled o základních i pokročilých metodách analýzy dat a základních i pokročilých aspektech jazyka Python, který se naučí používat ve spojení s moderními matematickými knihovnami a knihovnami pro pokročilou analýzu a modelování dat.
- Není cílem naučit programovat v jazyce Python či detailně seznámit se všemi aspekty jazyka Python, ale umožnit používat jazyk efektivně.
  - Důraz na pochopení úskalí interpretovaného jazyka a schopnosti se s nimi vypořádat.
  - Jak využít synergického efektu dynamicky typovaného jazyka a rychlého kódu v C
  - Pochopit, jakým způsobem fungují různé techniky implementované v různých knihovnách obecně a vědět, na jaká data je která technika vhodná.

Motivace

- Cíl: sečíst sekvenci hodnot (jakýkoliv objekt typu sekvence, prvky numerické hodnoty)

```
l = [i for i in range(100_000)]
```

```
def sum_numbers(lst : typing.Sequence[int]):
    sum = 0
    for item in lst:
        sum += item
    return sum

>>> timeit.timeit('f(l)', globals={'l':l,'f':sum_numbers}, number=1000)
3.8775279799010605 sec
```

```
>>> timeit.timeit('f(l)', globals={'l':l,'f':np.sum}, number=1000)
6.297680999850854 sec
```

```
>>> timeit.timeit('f(a(l))', globals={'l':l,'f':np.sum, 'a':np.array}, number=1000)
6.299477434018627 sec
```

```
>>> timeit.timeit('f(l)', globals={'l':l,'f':sum}, number=1000)
0.6946760187856853 sec
```

```
>>> timeit.timeit('f(l)', globals={'l':l,'f':sum_numbers_accelerated}, number=1000)
0.2992833990138024 sec
```

Náplň kurzu

- Osnova přednášek
  1. Úvod do jazyka I (vlastnosti a záludnosti jazyka)
  2. Úvod do jazyka II (pokročilé aspekty jazyka)
  3. Získávání dat a datová perzistence (získávání dat z web. zdrojů, serializace, komp. data, databáze)
  4. Základní přístupy k vizualizaci dat (úvod do principů knihovny matplotlib)
  5. Efektivní realizace operací nad n-dimenzionálními poli (úvod do numpy)
  6. Nástroje pro pokročilou manipulaci s daty (čas. řady, agregace, shlukování)
  7. Základní metody analýzy dat a datových závislostí (deskriptivní statistika, korelace, modelování)
  8. Pokročilé přístupy k vizualizaci dat (vztahy, histogramy, párové grafy, teplotní mapy)
  9. Práce s obrazovými daty a možnosti prezentace dat (obraz, PDF, DOC, XLS, ...)
  10. Operace nad časovými řadami a geografickými daty (podobnost, prediktivní analýza, dotazování)
  11. Pokročilé metody analýzy dat a datových závislostí (regresní analýza, interpolace dat)
  12. Výpočty v symbolické doméně (analytické řešení rovnic, dif. rovnice, Z3)
  13. Možnosti akcelerace kódu pro potřeby HPC (Cython, Numba)

Klasifikace

- Zakončení
  - klasifikovaný zápočet
- Podmínky udělení zápočtu
  - vypracování samostatného projektu a získání minimálně 50 bodů celkem a minimálně 2 body z každé části projektu
- Projekt
  - tvoří tři části – získání dat, základní vizualizace (20 b), pokročilejší vizualizace (20 b), statistické zpracování, generování zprávy (60 b)
  - části se odevzdávají průběžně – první 12.11.2021, druhá 10.12.2021, finální řešení 7.1.2022
- Bonusové úlohy
  - jednorázová dobrovolná a symbolicky hodnocená aktivita pro prohloubení znalosti

Interactive computing

Python umožňuje několik způsobů interaktivního výpočtu:

- Klasický Python shell.

```
$ python3.8
Python 3.8 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Interactive Python shell (apt install ipython3)

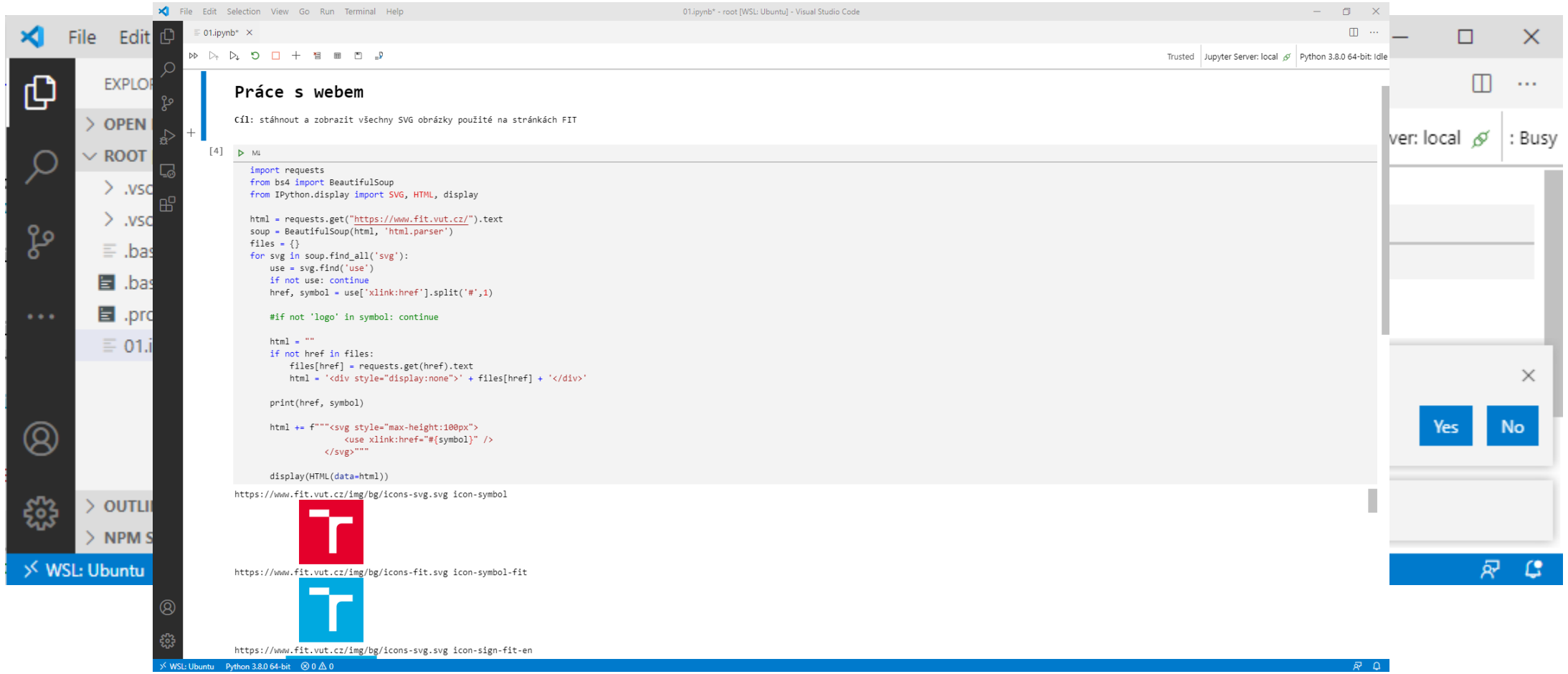
```
$ ipython3
…
IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: range?
Init signature: range(self, /, *args, **kwargs)
```

- Jupyter notebook (nativní podpora v multiplatformním Visual Studio Code)
  - umožňuje nativní vizualizaci grafiky, tabulek, HTML
  - možnost střídat úseky kódu s markdown dokumentací

Jupyter notebook ve Visual Studio Code

? Kolik typů buněk má Notebook?

- Jednotné, multiplatformní a snadno zprovoznitelné prostředí
  - lze pracovat lokálně na Windows, Linux nebo Windows WSL, Mac OS
  - lze pracovat vzdáleně na školních strojích (merlin.fit.vutbr.cz) z Windows, Linux nebo Mac OS



Instalace balíků v prostředí Visual Studio Code

- Možnosti instalace
  - přes terminál (je potřeba si pohlídat verzi Python)

```
> pip install matplotlib
```

  - v Jupyter notebook pomocí volání příkazů shellu

```
!pip install matplotlib
```

  - v Jupyter notebook pomocí tzv. line magic %pip (viz dokumentace ipython)

```
%pip install matplotlib
```

  - v Jupyter notebook programově pomocí python modulu pip

```
try:
    from pip import main as pipmain
except:
    from pip._internal import main as pipmain

pipmain(['install','matplotlib'])
```

Pokročilé aspekty jazyka Python

Zdeněk VAŠÍČEK
Fakulta Informačních Technologií, Vysoké Učení Technické v Brně
Brno, Czech Republic
vasicek@fit.vutbr.cz

Obsah

- Python jakožto interpretovaný objektově orientovaný dynamicky typovaný jazyk se silnou typovou kontrolou
  - Koncept objektů první kategorie (tzv. first-class objects)
  - Modifikovatelnost objektů (Mutable vs. Immutable objects)
- Objektová introspekce
- Základní nástroje pro profilování kódu
  - určení časové a paměťové náročnosti kódu
  - profilování kódu
- Reprezentace čísel
- Pokročilé aspekty jazyka
  - Uzávěr
  - Dekorátory
  - Iterátory

Vlastnosti jazyka Python

Aneb nahlédnutí pod kapotu …

Python jako programovací jazyk

- Python není ani čistě kompilovaný ani čistě interpretovaný jazyk
  - Kód (.py) se zpracovává řádek po řádku a vždy se vytváří tzv. byte kód.
  - Byte kód může být buď **interpretován** (CPython implementace) nebo **přeložen** JIT překladačem (PyPy implementace).
  - Jako kompilovaný jazyk se ukládá byte kód do souboru (.pyc) a nedojde-li ke změně, zdrojový kód se znovu nepřekládá (lze využít jak k tvorbě skriptů tak k tvorbě rozsáhlých aplikací).
- Nejrozšířenější CPython (stručně Python) je referenční implementace, ale existuje řada alternativních (a výkonnějších) implementací:
  - Cython - interpret Python rozšířeného o prvky jazyka C
  - PyPy – alternativní implementace Python využívající JIT překladu (implementace v restricted Python)
  - IronPython – implementace v jazyce .Net
  - Jython – implementace v jazyce Java
- Python virtual machine
  - základ hybridního interpretu Python, který využívá zásobníkového modelu (first-in last-out)

Objektová orientace

- Python je objektově orientovaný jazyk, kde každá entita je objektem
  - Každý objekt má vlastní identitu a lze se na něj odkazovat
  - Python objekty jsou tzv. objekty první kategorie (First-class object) => uniformní přístup
- First-class object lze vytvořit dynamicky a platí, že může být použit jako
  - hodnota proměnné
  - vstup funkce
  - návratová hodnota funkce
  - člen datových struktur
  - anonymní (nepojmenovaný)

```
def func(x):
    def inner(x):
        return x
    return inner
```

```
x = func
func([1,2,func])
func(func)
func(lambda x: x)
```

- Příklady entit, které mohou být předávány jako jakýkoliv jiný objekt
  - funkce první kategorie (first-class function) – viz uzávěr a lambda funkce
  - třídy první kategorie (first-class class) – viz metatřídy

Python z pohledu datových typů

- Python je dynamicky typovaný jazyk (tzv. dynamically typed language)
  - Typová kontrola je prováděna za běhu, nikoliv při kompilaci jako u staticky typovaných jazyků.
  - Každý objekt (nikoliv proměnná!) je spojen s nějakou strukturou nesoucí informaci o jeho typu.

```
x = 'string'
x = 50
x = lambda x: x
```

- Python je jazyk se silnou typovou kontrolou (tzv. strongly typed language)
  - Datový typ určitého objektu se sám od sebe nezmění, vždy je zapotřebí explicitní konverze (pozor však na výrazy, 1 + 1.0 je legální výraz typu float – viz dočasné kopie).

```
'1' + 3
TypeError: can only concatenate str (not "int") to str
```

? Proč je legální `'1' * 3`?

Objektová orientace

- Data jsou v Python programu reprezentována pomocí objektů a vztahů mezi objekty.
  - „proměnné“ jsou pouze ukazatel na Python objekt
  - potřeba GC (Garbage Collector), který sleduje, které objekty již nejsou referencovány a je možné uvolnit
- Každý objekt má
  - svou identitu
    - jednoznačný neměnný identifikátor, se kterým pracuje operátor **is**
  - asociován datový typ
    - datový typ je neměnný a definuje možné hodnoty a operace nad objektem
  - stav (hodnotu)
    - stav objektu se může měnit, avšak existují neměnné typy (tzv. Immutable), které toto nepřipouští
  - vlastní počítadlo referencí
    - viz správa paměti (smazat lze v Python pouze jméno, nikoliv objekt)

PyLongObject v CPython

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
```

viz longintrepr.h

```
#define PyObject_VAR_HEAD \
    PyVarObject ob_base;
```

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size;
} PyVarObject;
```

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;
```

Základní podpora objektové introspekce (Object introspection)

- Python podporuje introspekci (viz built-in funkce), u každého objektu lze zjistit
  - identitu

```
>>> x = 5
>>> id(x)
140726681405216
```

```
>>> x = "str"
>>> id(x)
1671772051888
```

```
>>> fun = lambda x: x
>>> id(fun)
1671869722480
```

  - typ

```
>>> x = 5
>>> print(type(x))
<class 'int'>
```

```
>>> x = 5.0
>>> print(type(x))
<class 'float'>
```

```
>>> fun = lambda x: x
>>> print(type(fun))
<class 'function'>
```

  - seznam atributů a metod objektů

```
assert isinstance(int, type) == True
dir(int)
['__abs__', '__add__', '__and__', ..., 'imag', 'numerator', 'real', 'to_bytes']
```

  - stav počítadla referencí (funkce *getrefcount* z modulu sys)

```
>>> sys.getrefcount(5000)
3
```

Objekty z pohledu možnosti změny stavu (Mutable vs Immutable Objects)

- Python rozlišuje mezi tzv. modifikovatelnými (Mutable) a neměnnými (Immutable) objekty

Skupina	Datový typ	Typ
Řetězec (Text Type):	str	Immutable
Číslo (Numeric Types):	int, float, complex	Immutable
Sekvence (Sequence Types):	list	Mutable
	tuple, range	Immutable
Slovník (Mapping Type):	dict	Mutable
Množina (Set Types):	set	Mutable
	frozenset	Immutable
Boolovská hodnota (Boolean Type):	bool	Immutable

? Jak se chová int při inkrementaci hodnoty?

? Jak zjistit o jaký typ objektu se jedná?

- Neměnné objekty jsou dražší z pohledu manipulace (nová instance) avšak výhodnější z pohledu předávání dat (předávání odkazem => přiřazení nikdy nekopíruje data)

Záludnosti modifikovatelných a neměnných objektů

- Přiřazení nikdy nekopíruje data
  - jednoduché přiřazení

```
nums = [1, 2, 3]
x = nums
nums.append(4)
```

? Jaký je obsah x?

  - proměnné *for* cyklu se též přiřazuje objekt odkazem

```
nums = [1, 2, 3]
for i, x in enumerate(nums):
    print(i, id(nums[i]), id(x))
    x += 1
assert nums == [2,3,4]
```

? Proč je vyvolána výjimka?

  - argumentu funkce se při volání taktéž přiřazuje objekt odkazem

```
def capitalize(wlist):
    for i, w in enumerate(wlist):
        wlist[i] = w.capitalize()
    return wlist
words = ["python","word"]
capitalized = capitalize(words)
```

? Jaký obsah má words po zavolání funkce?

Záludnosti modifikovatelných a neměnných objektů

? Opravdu někde došlo ke kopii?

- Dočasné kopie (temporary copy)
  - Ve výrazech dochází implicitně k vytváření dočasných kopií objektů aby nedocházelo k side efektům.
  - ! pozor na rozdíl ve výkonnosti in-place operátorů (např. +=) a běžných binárních operátorů (např. +)

```
a = [1, 2, 3]
a = a + (a + [4])
```

vs.

```
a = [1, 2, 3]
a += a
a += [4]
```

vs.

```
a = [1, 2, 3]
a.extend(a)
a.append(4)
```

- Rozbalení (sequence/iterable unpacking)
  - takto se poslední prvek sekvence opravdu nezjišťuje (viz iterátor)

```
L = [i for i in range(10000)]
*_,a = L
```

- Efektivní vykonávání kódu je typicky na úkor čitelnosti
  - typická vlastnost i u pokročilých knihoven jako je např. numpy

Záludnosti modifikovatelných a neměnných objektů

- Neměnnost n-tice neznamená nemožnost změny obsahu položky

```
tuple = (1, [2])
>>> tuple[1]
[1]
>>> tuple[1]=[1]    TypeError: 'tuple' object does not support
item assignment
```

```
tuple.__getitem__(1)

tuple.__setitem__(1, [1])
```

  - je-li položka modifikovatelná, lze ji změnit

```
>>> tuple[1].clear()
>>> tuple[1].append(1)
>>> tuple
    (1, [1])
```

Záludnosti modifikovatelných a neměnných objektů

- Podivné vyvolání výjimky v případě aktualizace seznamu v n-tici
  - n-tice je neměnná, avšak modifikovat obsah seznamu musí být možné (využijeme += odpovídající u seznamu volání metody **extend**)

```
>>> tuple = (1,[2])
>>> tuple[1] += [3]
TypeError: 'tuple' object does not support item assignment
```

? Co způsobilo výjimku?

  - došlo sice k výjimce, obsah však byl aktualizován

```
>>> tuple
(1, [2, 3])
```

```
>>> tuple = (1,[2])
>>> list = tuple[1]
>>> list += [3]
```

Další možnosti objektové introspekce

- Funkce jsou taktéž objektem a dovolují získat informace ze speciálního atributu __code__ (tzv. Code Object) a to až na úroveň Byte code

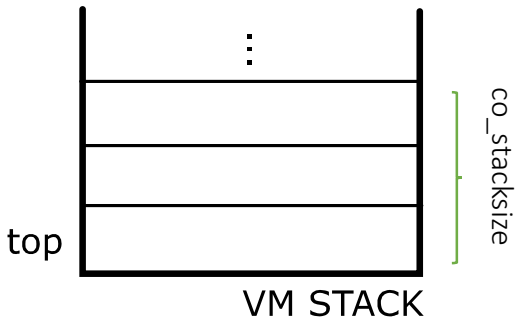
```
def addsuffix(a, n=1):
    return a + "suffix"*n

>>> code_obj = addsuffix.__code__
>>> code_obj.co_argcount
    2
>>> code_obj.co_name
    'fun'
>>> code_obj.co_varnames
    ('a', 'n')
>>> code_obj.co_consts
    (None, 'suffix')
>>> code_obj.co_code
    b'|\x00d\x01t\x00\x14\x00\x17\x00S\x00'
```

více viz dokumentace docs.python.org/3/library/inspect.html

Další možnosti objektové introspekce, princip VM

- Byte code je možné disassemblovat (převést na posloupnost instrukcí pro Python VM) s využitím nástrojů modulu dis

```
def addsuffix(a, n=1):
    return a + "suffix"*n

>>> import dis
>>> code_obj = addsuffix.__code__
>>> dis.dis(code_obj.co_code)
…
>>> dis.dis(addsuffix)
 0 LOAD_FAST                0 (a)
 2 LOAD_CONST               1 ('suffix')
 4 LOAD_FAST                1 (n)
 6 BINARY_MULTIPLY
 8 BINARY_ADD
10 RETURN_VALUE

>>> code_obj.co_stacksize
3
```



detaily viz dokumentace docs.python.org/3/library/dis.html

Další možnosti objektové introspekce

- Řadu informací o objektech lze získat prostřednictvím funkcí modulu inspect
  - funkce pro testování typu objektu
    ismodule, isclass, ismethod, isfunction, isgeneratorfunction, isgenerator, isbuiltin, iscode, …
  - funkce týkající se zdrojového kódu
    getfile, getmodule, getsourcelines, …
  - funkce getmembers vrací všechny vnitřní atributy objektu
  - kompletní přehled viz dokumentace docs.python.org/3/library/inspect.html
- Příklad zjištění signatury deklarované funkce:

```
def fun(a, *, b:int, **kwargs):
    pass

fun(1,b=1)
```

```
from inspect import signature
sig = signature(fun)
str(sig) # (a, *, b: int, **kwargs)
len(sig.parameters)  # 3
sig.parameters['b'] # <Parameter "b: int">
```

Nevýhoda objektového přístupu, dynamického typování a interpretace kódu

- Dynamické typování znamená zvýšenou režii
  - Interpret nezná předem typy proměnných (nejsou fixní)
  - Statická analýza je obtížná

```
def add(a, b=1):
    return a + b
```

- Objektový model vede typicky k neefektivnímu způsobu přístupu do paměti, velké režii
  - Immutable typy (např. čísla) sice usnadňují práci, ale znamenají potřebu neustálé alokace; sekvence int zabírá 10x více paměti než pole v C
  - Pole hodnot často netvoří souvislý blok v paměti (nelze využít CPU cache)
- Interpretace kódu vnáší sama o sobě režii
  - pro interpret je obtížné provádět optimalizace běžné u kompilovaných jazyků (detekovat neužitečné operace, opakující se přístupy, …)

Popularita jazyka Python

- Python je velmi populární jazyk
  - druhý nejoblíbenější / nejpoužívanější jazyk v open-source viz madnight.github.io/githut
  - první v trendech vyhledávání pypl.github.io/PYPL.html
  - oblíbený v mnoha oblastech – vědecko-technické výpočty, skripty, webové služby, …
- Důvod #1: Dynamické typování dělá Python jednodušším z pohledu použití oproti C/C++
  - kratší kód, jednodušší deklarace, čitelnější kód, bezproblémové předávání objektů mezi funkcemi, jednodušší implementace polymorfismu
  - možnost interaktivní práce
- Důvod #2: Python je velice flexibilní, má řadu zkratek podporujících rychlou tvorbu kódu
  - jednoduchá syntaxe s minimem problematických konstrukcí (zaměření na čitelnost kódu), jednodušší testování (lze udělat náhražku čehokoliv), možnost interakce s nízko úrovňovými knihovnami (DLL, SO)
- Důvod #3: Python nabízí možnost, jak se vypořádát s neefektivitou, je-li to zapotřebí
  - modul v C/C++ (neefektivní z pohledu programátorského usílí), viz **numpy**, **scipy**, …
  - anotace zdrojového kódu datovými typy (často přímočaré a málo pracné), viz. **cython**
  - využití JIT překladačů, viz např. **numba**, **numexpr**, …

Souhrn

- Každá entita v Python je objekt
  - data jsou reprezentována objekty a vztahy mezi nimi
- Každý objekt má identitu, typ, hodnotu
  - identita je neměnná po celou dobu životnosti objektu
  - typ definuje chování objektu a jakých může nabývat hodnot
- V Python existují dva typy objektů z pohledu změny hodnoty
  - Mutable vs. Immutable
- Python chápe proměnné odlišně než kompilované jazyky
  - Python má názvy odkazující na objekty referencemi
- Objektový přístup a dynamické typování představuje režii
  - vyšší paměťové nároky, pomalejší zpracování
- Existují prostředky, jak redukovat jak paměťové, tak časové nároky
  - modul v C, statické typování, …

Nástroje pro analýzu paměťové a časové náročnosti kódu

Profilování kódu

Nástroje pro profilování kódu

- V případě zpracování objemných dat typicky narazíme v průběhu tvorby kódu na problém s rychlostí kódu nebo paměťovými nároky kódu
- Přehled základních možností pro stanovení nároků a profilování kódu:

Modul	**Popis**	**iPython/Jupyter**
timeit (vestavěný)	měření doby vykonání kódu	%time, %timeit
sys (vestavěný)	getsizeof vracející počet bytů zabraných objektem	
pympler (externí)	asizeof – deep varianta getsizeof	
profile (vestavěný)	profilování kódu s rozlišením na úrověň funkcí	%prun
line_profiler (externí C)	profilování kódu s rozlišením na úrověň řádků	%lprun
memory_profiler (externí C)	měření spotřebované paměti	%memit, %mprun
tracemalloc (vestavěný)	pořizování snapshotů alokované paměti	
filprofiler (externí C)	peak memory usage	%%filprofile

Měření doby vykonávání kódu

- Modul timeit nabízí několik funkcí pro měření doby vykonávání kódu
  - *timeit* vytvoří instanci Timer, jedenkrát provede setup a number-krát stmt

```
timeit.timeit(stmt='pass', setup='pass', number=1000000, globals=None)
```

  - *repeat* vytvoří instanci Timer a spustí repeat-krát timeit, vrací repeat výsledků

```
timeit.repeat(stmt='pass', setup='pass', repeat=5, number=1000000, globals=None)
```

  - více viz dokumentace docs.python.org/3/library/timeit.html
- Funkce modulu timeit
  - lze volat přímo, nebo

```
>>> timeit.timeit('f(l)', globals={'l':l,'f':sum_numbers}, number=1000)
    3.8775279799010605 sec
```

  - využít tzv. magic commands v iPython/Jupyter

Měření doby vykonávání kódu – iPython magics

- Doba vykonávání jednoho spuštění (%time)

```
>>> L = [abs(-i+500000) for i in range(1000000)]
>>> %time L.sort()
    Wall time: 12 ms
>>> L.sort()
>>> %time L.sort()
    Wall time: 5.99 ms
```

- Doba vykonávání vypočítaná z vícenásobného spuštění (%timeit)
  - počet opakování je stanoven automaticky (nebo lze definovat parametrem), lze specifikovat setup

```
>>> L = [abs(-i+500000) for i in range(1000000)]
>>> %timeit L.sort()
    5.67 ms ± 182 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Doba vykonávání celé buňky (%%time, %%timeit)
  - pozn. na prvním řádku %%timeit je setup

```
%%timeit -n 5 -r 7 L = [abs(-i+500000) for i in range(1000000)] #setup code
L.sort()
```

```
7.24 ms ± 627 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

Množství alokované paměti

- Množství alokované paměti určitým objektem lze zjistit
  - pomocí funkce getsizeof z vestavěného modulu sys (! vrací defacto pole ob_size z PyObject)

```
>>> obj = [1, 2, (3, 4), 'abcd']
>>> sys.getsizeof(obj)
    88 # 56 + 8*pocet polozek
>>> sys.getsizeof(1234)
    28
```

  - pomocí funkce asizeof z externího moduly pympler (velikost včetně všech referencovaných objektů)

```
>>> obj = [1, 2, (3, 4),  'abcd']
>>> from pympler import asizeof
>>> asizeof.asizeof(obj)
    328
>>> asizeof.asized(obj, detail=1).format()
    [1, 2, (3, 4),'abcd']  size=328 flat=88
    (3, 4)                 size=120 flat=56
    'abcd'                 size=56  flat=56
    1                      size=32  flat=32
    2                      size=32  flat=32
```

Profilování kódu – časová náročnost

- Profilování pomocí interního profileru s rozlišením na úroveň funkcí
  - lze pomocí magic %prun

```
def checksum(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total

%prun checksum(1000000)
```

```
14 function calls in 0.879 seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.726    0.145    0.726    0.145 <ipython-input-10-e23e14e75fa4>:4(<listcomp>)
        5    0.109    0.022    0.109    0.022 {built-in method builtins.sum}
        1    0.034    0.034    0.869    0.869 <ipython-input-10-e23e14e75fa4>:1(checksum)
        1    0.010    0.010    0.879    0.879 <string>:1(<module>)
        1    0.000    0.000    0.879    0.879 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Profilování kódu – časová náročnost

- Profilování pomocí profileru s rozlišením na úroveň řádků (externí C modul)
  - lze pomocí magic %lprun (nutno mít načtený modul line_profiler pomocí load_ext)

```
def checksum(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total

%load_ext line_profiler
%lprun -f checksum checksum(1000000)
```

```
Total time: 1.20464 s
Function: checksum at line 1
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def checksum(N):
     2         1          3.0      3.0      0.0      total = 0
     3         6         12.0      2.0      0.0      for i in range(5):
     4         5    1168642.0 233728.4     97.0          L = [j ^ (j >> i) for j in range(N)]
     5         5      35984.0   7196.8      3.0          total += sum(L)
     6         1          1.0      1.0      0.0      return total
```

Profilování kódu – paměťová náročnost

- Profilování pomocí profileru s rozlišením na úroveň funkcí (externí C modul)
  - lze pomocí magic %memit (nutno mít načtený modul memory_profiler pomocí load_ext)

```
def checksum(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total

%load_ext memory_profiler
%memit checksum(1000000)
```

```
peak memory: 148.58 MiB, increment: 70.30 MiB
```

Graf relací mezi objekty

%pip install objgraph

- Externí modul objgraph umožňuje graficky vizualizovat vztahy mezi objekty
  - dopředné reference

```
import objgraph
x = []
y = [x, [x], dict(x=x)]
objgraph.show_refs([y])
```

  - zpětné reference

```
import objgraph
x = []
y = [x, [x], dict(x=x)]
print(sys.getrefcount(x)-1) # 4
objgraph.show_backrefs([x])
```

? Jak vypadájí zpětné odkazy u celých čísel, např. -5?

- Graf závislostí je užitečný v případě, kdy dochází k neustálému nárůstu množství alokované paměti
  - více viz např. investigating-memory-usage-in-python-program

Byte kód jako nástroj pro pochopení optimalizací v programu

- Rozbalení smyček
  - Na první pohled by se mohlo zdát, že použití range ve for cyklech implikující práci s iterátory bude vnášet značnou režii. V praxi tomu tak však není, rozbalení smyček je významně horší, proč?

```
def first(N=10000):
    for i in range(N):
        pass
```

```
def second(N=10000):
    i = 0
    while i < N:
        i += 1
```

```
%timeit first()
#208 µs ± 17.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit second()
#546 µs ± 44.8 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Byte kód jako nástroj pro pochopení optimalizací v programu

- Resoluce jmen v dynamickém jazyce
  - Resoluce jmen ze své podstaty musí probíhat dynamicky. Víme-li, že v kódu přistupujeme k témuž vzdálenému objektu, pak jeho caching v lokální proměnné významně zrychlí provádění kódu.

```
def first(N=1000):
  total = 0
  for x in range(N):
    total += math.sin(x)
  return total
```

```
def second(N=1000):
  sin = math.sin
  total = 0
  for x in range(N):
    total += sin(x)
  return total
```

```
%timeit first()
#233 µs ± 35.6 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit second()
#172 µs ± 16.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Reprezentace čísel

Číselné datové typy

Manipulace s velkými celými čísly

- V prostředí Python jsme schopni pracovat nativně s celými čísly aniž bychom museli specifikovat bitovou šířku
  - Řada programovacích jazyků typicky rozlišuje mezi signed / unsigned verzí celých čísel a bitovou šířkou

```
>>> len(str(10**12345)) #celé číslo mající 12346 číslic
12346
>>> len(str(int('1100'*1024,2))) # binární hodnota délky 4x1024 bitů
1233
```

- Python (C implementace) nativně pro uchování celých čísel používá
  - buď 4 bytovou (32-bit Python) nebo 8 bytovou (64-bit Python) reprezentaci pro malé hodnoty, nebo
  - sekvenci 15-bitových (32-bit Python) nebo 30-bitových (64-bit Python) hodnot, tzv. digits, která se používá pro reprezentaci hodnot, které není možné mapovat do předchozího typu

- Rozsahy reprezentací
  - Informaci o maximální hodnotě pro první typ lze zjistit pomocí konstanty sys.maxsize,
  - informaci o druhém způsobu reprezentace pomocí sys.int_info

Práce s racionálními a iracionálními čísly

- Iracionální čísla
  - Reálná čísla, která nelze zapsat ve tvaru zlomku (tj. ve tvaru podílu dvou celých čísel).
  - V desetinném zápise by měla nekonečný desetinný rozvoj bez periody.
- Číslo s plovoucí řádovou čárkou typu float (IEEE DP) vždy pouze aproximuje hodnotu
  - Možný zdroj chyb, viz:

```
>>> priceDiff = 219.92 - 219.52 #219.92 - 219.52 = 0.40
>>> print(">=0.4" if priceDiff >= .40 else "<0.4")
"<0.4"
```

```
>>> (0.1 + 0.1 - 0.3 + 0.1) == 0
False
```

  - Místo operátoru = je zapotřebí typicky použít ε test

```
>>> print(">=0.4" if (priceDiff-0.4)<1e-10 else "<0.4")
">=0.4"
```

```
>>> abs(0.1 + 0.1 - 0.3 + 0.1) < 1e-15
True
```

- Podpora v Python pro přesnou práci s racionálními a iracionálními čísly:
  - vestavěný modul **fractions** obsahující třídu **Fraction** pro práci s reálnými čísly ve tvaru zlomku
  - vestavěný modul **decimal** obsahující třídu **Decimal** pro přesnou práci s racionálními čísly (čísly s řádovou čárkou), typicky ve finančním sektoru

Práce s racionálními a iracionálními čísly

- Decimal
  - implementuje standard Decimal arithmetic ANSI standard X3.274-1996
  - interně aritmetika s omezenou avšak mnohem vyšší přesností než IEEE DP (viz PEP-0327)

```
>>> (Decimal('0.1') + Decimal('0.1') - Decimal('0.3') + Decimal('0.1')) == 0
True
```

Příklad:

```
>>> from decimal import Decimal
>>> tax_rate = Decimal('17')/Decimal(100)
>>> purchase_amount = Decimal('2.93')
>>> total = tax_rate * purchase_amount
Decimal('0.4981')
```

```
>>> tax_rate = 17.0/100
>>> purchase_amount = 2.93
>>> total = tax_rate * purchase_amount
0.49810000000000004
```

zaokrouhlení na setiny:

```
>>> penny=Decimal('0.01')
>>> total.quantize(penny)
Decimal('0.50')
```

```
>>> round(total*100)/100.0
0.5
```

Práce s racionálními a iracionálními čísly

- Fraction
  - Některá čísla není možné reprezentovat konečným desetinným rozvojem, např. 1/3=0.33333333

```
>>> x = Decimal(1)/Decimal(3) - Decimal(2)/Decimal(3) + Decimal(1)/Decimal(3)
>>> print(repr(x))
Decimal('-1E-28')
>>> x == 0
False
```

  - a je tudíž nutné v případě potřeby použít reprezentaci pomocí zlomků

```
>>> from fractions import Fraction
>>> x = Fraction(1,3) - Fraction(2,3) + Fraction(1,3)
>>> print(repr(x))
Fraction(0, 1)
>>> x == 0
True
```