Úvod do programování v jazyce Python

Jan Pytel

České vysoké učení technické

11. května 2024

Struktura přednášek I

Struktura přednášek II

Pravidla předmětu

Programování je zábava, přestože první kroky bývají obtížné.
Ambice je vás uvést do skvělého světa programování.

- Kontakt `jan.pytel@gmail.com`
- Cvičení jsou nepovinná
- Striktní rozdělení přednášek a cvičení (změna oproti *Úvod do SQL*)
- Zápočet obdrží každý automaticky
- Během cvičení bude 5 úkolů - prvních úspěšných 5 řešitelů obdrží 10 bodů, zbylých 10 obdrží 5 bodů.
- V případě dosažených bodů během semestru:

  46 - garantovaná známka **B**
  36 - 45 - garantovaná známka **C**
  26 - 35 - garantovaná známka **D**
  20 - 25 - garantovaná známka **E**

Přednáška 1

Úvod do programování - problém, algoritmus, základní vlastnosti algoritmu

"A computer program does what you tell it to do, not what you want it to do."- Unknown

Co je to programování

Programování je proces vytváření instrukcí, které počítač provede k řešení určitého problému.

- V programování se problémem rozumí úkol nebo výzva, kterou je třeba řešit pomocí počítačového programu.
- Porozumění problému je prvním krokem v procesu vývoje softwaru.
- Dáme si za úkol prozkoumat koncept problémů v programování a jak jsou řešeny.

Problém

Problém - identifikace

- Prvním krokem při řešení problému je jeho jasná definice a porozumění.
- To zahrnuje rozložení problému na menší části a identifikaci požadavků.
- Správná identifikace problému je klíčová pro vytvoření efektivního řešení.

Problém - analýza

- Jakmile je problém identifikován, je důležité analyzovat jeho složitost a omezení.
- Porozumění vstupů, výstupů a omezení problému je zásadní pro návrh vhodného algoritmu.
- Analýza problému pomáhá určit nejefektivnější a nejúčinnější přístup k jeho řešení.

Problém - řešení

- Řešení programového problému zahrnuje návrh algoritmu nebo sady instrukcí k dosažení požadovaného výsledku.
- Tento algoritmus by měl být logický, efektivní a schopný zpracovat různé scénáře.
- Testování a doladění řešení je iterativní proces, který zajišťuje jeho správnost a účinnost.

Algoritmy

Algoritmy

- **Algoritmus**: Algoritmus je krok za krokem postupující postup pro řešení určitého problému.
- **Vlastnosti algoritmů**:
  - **Efektivnost**: Schopnost algoritmu provádět úkol rychle a s minimálním využitím prostředků.
  - **Konečnost**: Algoritmus by měl skončit po konečném počtu kroků.
  - **Konečný vstup a výstup**: Algoritmus má definovaný vstup a výstup.
  - **Jednoznačnost**: Každý krok algoritmu musí být jasně definován a nemůže být víceznačný.
  - **Obecnost**: Algoritmus by měl být schopen řešit obecný problém, ne jen konkrétní instanci.

Vlastnosti Algoritmů (Pokračování)

- **Determinovanost**: Každý krok algoritmu musí být definován jednoznačně a předvídatelně.
- **Optimalita**: Algoritmus by měl řešit problém s minimálním možným počtem kroků.
- **Rekursivita**: Některé algoritmy mohou být definovány pomocí sebe sama.
- **Důslednost**: Algoritmus by měl dávat stejný výsledek pro stejný vstup.

Příklad algoritmu - výpočet lichých čísel

- **Název Algoritmu**: LichaCislaVypocet
- **Vstup**: Žádný
- **Výstup**: Seznam lichých čísel od 1 do 1000

Výpočet lichých čísel - algoritmus v Pseudokódu

for $i \leftarrow 1$ **do** 1000 **do**
    if i **je liché then**
        Přidej i do seznamu lichých čísel
    end if
end for

Výpočet lichých čísel - popis algoritmu

- Začněte s číslem 1.
- Pro každé číslo od 1 do 1000 zkontrolujte, zda je liché.
- Pokud je číslo liché, přidejte ho do seznamu lichých čísel.
- Opakujte tento proces pro všechna čísla od 1 do 1000.

Výpočet lichých čísel - C+ a Python

C++

```cpp
#include<iostream>

int main() {
  for (int i = 1; i <= 1000; i++) {
    if (i % 2 == 1) {
      std::cout << i << std::endl;
    } // if
  } // for
}
```

Python

```python
for i in range(1, 1000):
  if i % 2 == 1:
    print(i)
```

Výpočet lichých čísel do pole - C++

```cpp
#include<iostream>
#include<vector>

std::vector<int>  generuj_licha_cisla() {
  std::vector<int> vec;
  for (int i = 1; i <= 1000; i++) {
    if (i % 2 == 1) {
      vec.push_back(i);
    } // if
  } // for

  return vec;
} // generuj_licha_cisla_do_pole

int main() {
  for (int liche_cislo : generuj_licha_cisla()) {
    std::cout << liche_cislo << ", ";
  } // for
  std::cout << std::endl;
}
```

Výpočet lichých čísel do pole - Python

```python
def generuj_licha_cisla():
    licha_cisla = []
    for i in range(1, 1000):
        if i % 2 != 0:
            licha_cisla.append(i)
    return licha_cisla

print(generuj_licha_cisla())
```

Příklad algoritmu - seznam prvočísel od 1 do 1000

- **Název Algoritmu**: PrvocislaCalculator
- **Vstup**: Žádný
- **Výstup**: Seznam prvočísel od 1 do 1000

Výpočet prvočísel od 1 do 1000 - algoritmus v Pseudokódu

for $i \leftarrow 2$ to 1000 **do**
    $isPrime \leftarrow true$
    for $j \leftarrow 2$ to $\sqrt{i}$ **do**
        if $i \bmod j = 0$ **then**
            $isPrime \leftarrow false$
            break
        end if
    end for
    if $isPrime$ **then**
        Přidej i do listu prvočísel
    end if
end for

Výpočet Prvočísel

- Začněte s číslem 2.
- Projděte všechna čísla od 2 do 1000.
- Pro každé číslo zkontrolujte, zda je prvočíslo.
- Prvočísla jsou čísla dělitelná pouze jedničkou a samy sebou.
- Přidejte prvočísla do seznamu prvočísel.

Výpočet prvočísel od 1 do 1000 - Python

```
1  import math
2
3  # Python program ktery generuje list prvocisel od  1 do 1000
4  # Funkce vraci pro zadane cislo zdali se jedna o prvocislo
5  def je_prvocislo(num):
6      if num < 2:
7          return False
8      for i in range(2, int(math.sqrt(num))):
9          if num % i == 0:
10             return False
11     return True
12
13
14 for cislo in range(1, 1001):
15         if (je_prvocislo(cislo)):
16                 print(cislo)
```

Historie programovacích jazyků

Začátky programování

- **1936**: Konceptuální základy programování byly položeny Alanem Turingem s jeho myšlenkovým modelem - Turingovým strojem.
- **40tá léta**: První elektronické počítače, jako např. ENIAC a UNIVAC, vyvinuty v USA.
- **50tá léta**: Byl představen jazyk symbolických instrukcí (Assembly language), který umožňoval programátorům psát kód pomocí mnemotechnických instrukcí.
- **50tá léta**: Vznik prvních programovacích jazyků, včetně Fortranu, COBOLu a LISPu. Tyto jazyky programování lidem zpřístupnily.
- **60tá léta**: Vznik jazyka BASIC, který byl zaměřen na vzdělávání a začátečníky.

Rozvoj programování

- **70tá léta**: Vznik jazyka C, který se stal základem pro mnoho dalších jazyků.
- **80tá léta**: Vznik jazyka C++, který přinesl objektově orientované programování do popředí.
- **90tá léta**: Rozvoj moderních jazyků, jako např. Python, JavaScript, Ruby, které pokračují v inovaci a diversifikaci programovacích jazyků.
- **2000**-: Python se stal univerzálním a snadno naučitelným jazykem, široce používaným v různých oborech.
- Rovněž rozmach open-source a komunitního přístupu k vývoji a sdílení kódu.

Python

Guido Van Rossum holandský programátor, tvůrce programovacího jazyka Python. Narodil se 31. ledna 1956 v Haagu, Nizozemsko. Van Rossum vytvořil Python v roce 1989 a od té doby se stal jedním z nejpoužívanějších a oblíbených programovacích jazyků po celém světě. Guido van Rossum je známý svým přístupem k programování, který zdůrazňuje čitelnost a jednoduchost kódu.

Co je Python?

- Python je vysokoúrovňový programovací jazyk
- První vydání jazyka 1991 (Guido van Rossum)
- Jednoduchá a elegantní syntaxe, snadné čtení a psaní kódu.
- Mnoho knihoven a frameworků — jeden z nejoblíbenějších programovacích jazyků.
- Python je interpretovaný jazyk, což znamená, že kód je spouštěn přímo interpretrem, což usnadňuje rychlý vývoj a testování programů.
- http://www.python.org

Kde se Python používá

Jednoduchá odpověď: všude kde není potřeba low-level systém programování, real-time aplikace, ...

- Vývoj webových aplikací
- Data Science a Machine Learning
- Vědecké výpočty
- Automatizace a skriptování
- Vývoj her
- Desktopové aplikace
- Síťové technologie a kybernetická bezpečnost
- Finance a obchodování
- Internet věcí (IoT)

Proč Python *(a ne C++)?*

- Python je vhodný pro začátečníky - jednoduchá a čitelná syntaxe.
- Křivka učení
- Všestrannost
- Rozsáhlé knihovny
- Široká komunita

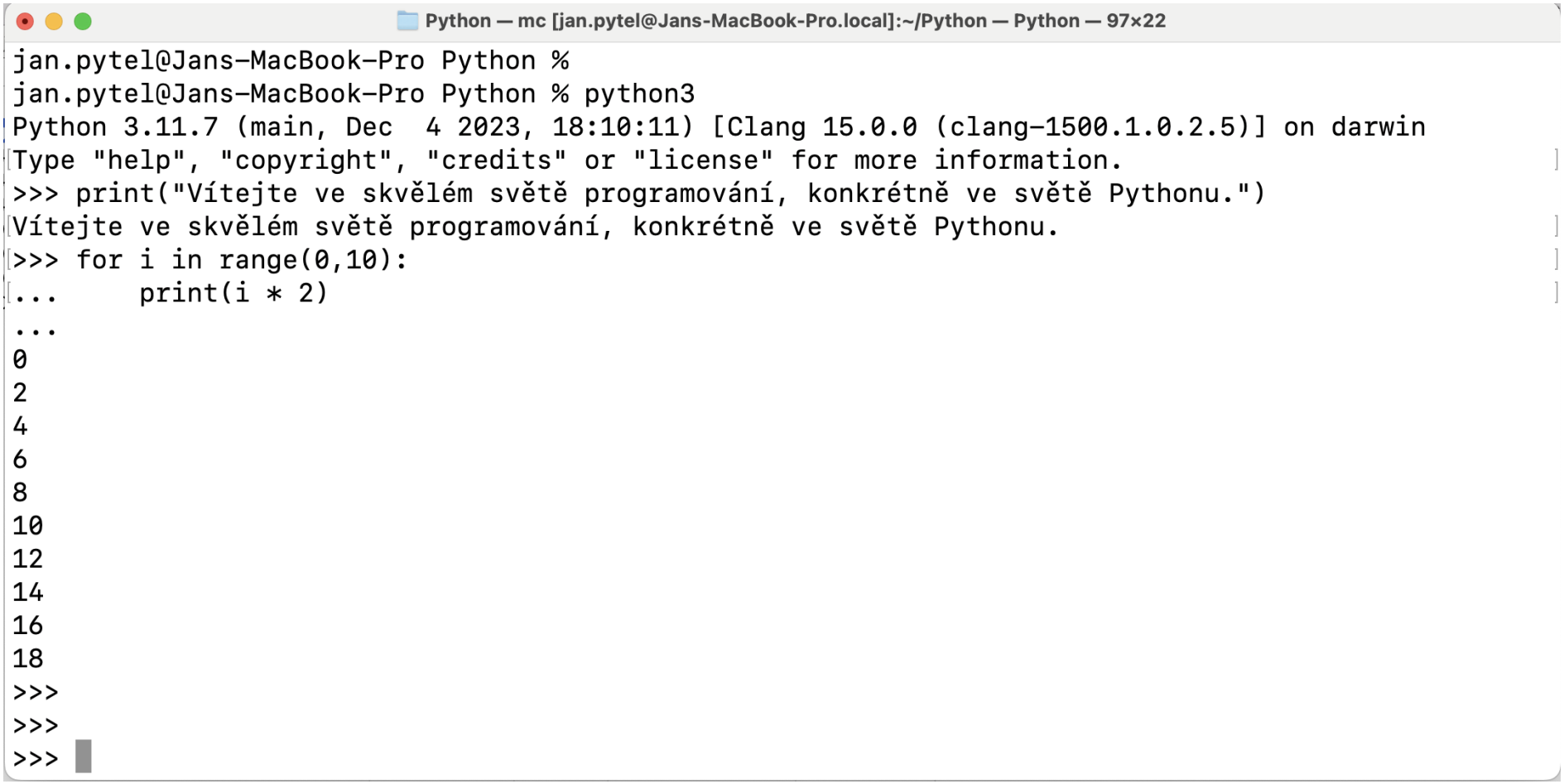
První aplikace v jazyce Python

Co je nutné pro spouštění první aplikace

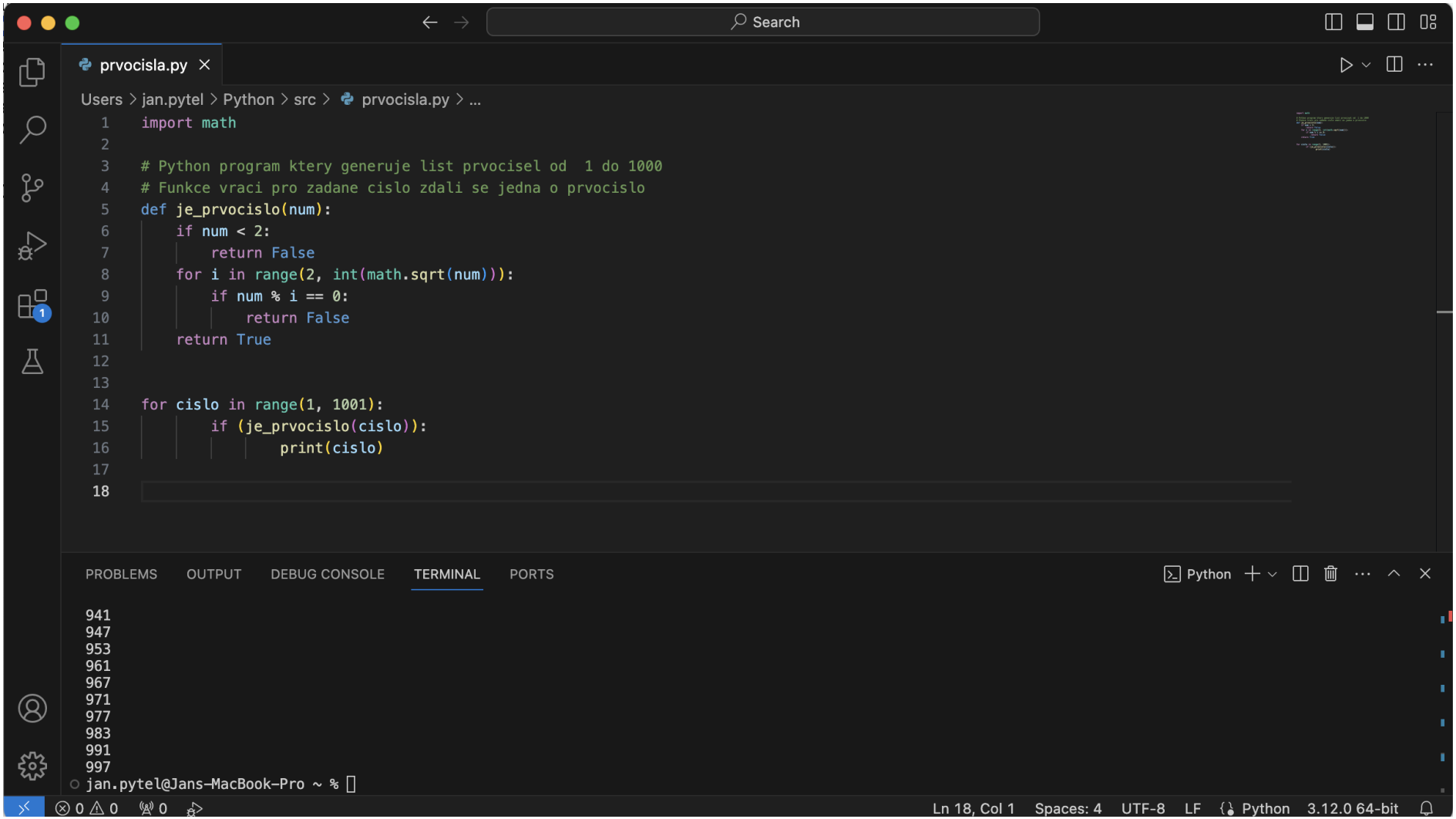
Před tvorbou první aplikace je nezbytné mít k dispozici alespoň jednu z následujících možností:

- **Práce v terminálu** - stažení jazyka Python z https://www.python.org/downloads/, pracujeme s versí $\geqslant 3.0.0$
- **Replit** - http://www.replit.com
- **Visual Studio Code** - https://code.visualstudio.com/
- **PyCharm** - https://www.jetbrains.com/pycharm/
- **Jupyterm** - https://pypi.org/project/jupyterm/
- a mnoho dalších ...

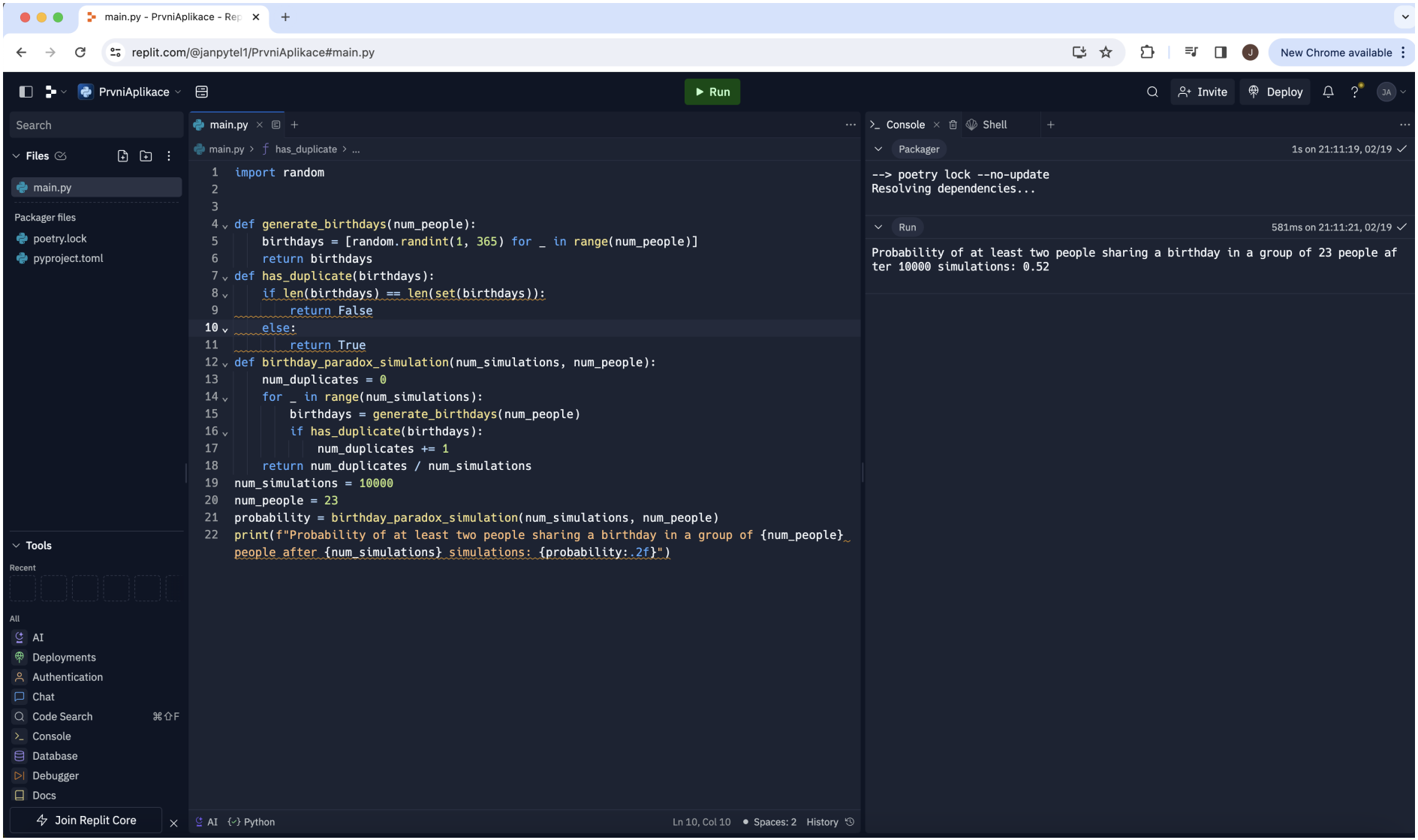
Práce v terminálu - interpret python3



Visual Studio Code



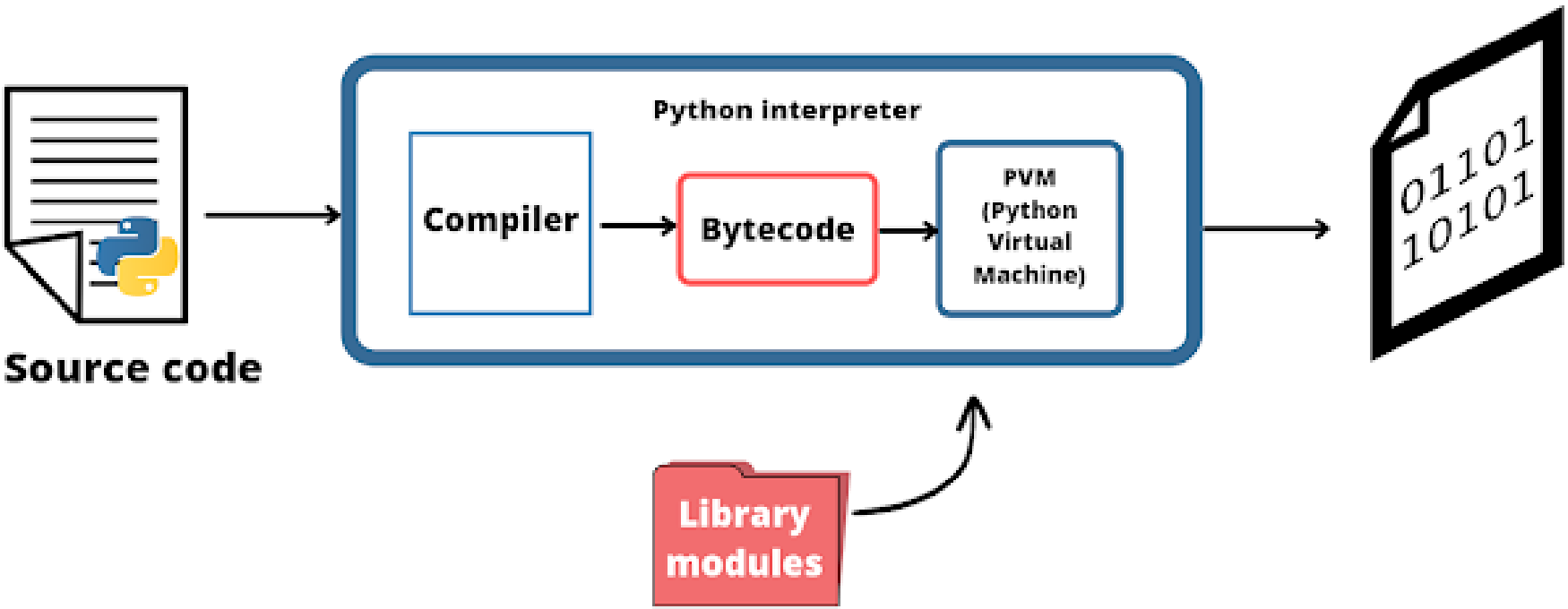
Replit


main.py - PrvniAplikace - Rep
replit.com/@janpytel1/PrvniAplikace#main.py
New Chrome available
PrvniAplikace
Run
Invite
Deploy
Search
Files
main.py
Packager files
poetry.lock
pyproject.toml
Tools
Recent
All
AI
Deployments
Authentication
Chat
Code Search ⌘⇧F
Console
Database
Debugger
Docs
Join Replit Core
main.py
main.py > has_duplicate > ...
import random
def generate_birthdays(num_people):
birthdays = [random.randint(1, 365) for _ in range(num_people)]
return birthdays
def has_duplicate(birthdays):
if len(birthdays) == len(set(birthdays)):
return False
else:
return True
def birthday_paradox_simulation(num_simulations, num_people):
num_duplicates = 0
for _ in range(num_simulations):
birthdays = generate_birthdays(num_people)
if has_duplicate(birthdays):
num_duplicates += 1
return num_duplicates / num_simulations
num_simulations = 10000
num_people = 23
probability = birthday_paradox_simulation(num_simulations, num_people)
print(f"Probability of at least two people sharing a birthday in a group of {num_people} people after {num_simulations} simulations: {probability:.2f}")
AI
Python
Ln 10, Col 10
Spaces: 2
History
Console
Shell
Packager
1s on 21:11:19, 02/19
--> poetry lock --no-update
Resolving dependencies...
Run
581ms on 21:11:21, 02/19
Probability of at least two people sharing a birthday in a group of 23 people after 10000 simulations: 0.52

Jak Python interně pracuje

Jak Python interně pracuje

Převzato z https://www.pycodemates.com/2022/01/how-python-works-is-it-really-an-interpreted-language.html



Kompilátor

Python kompilátor je komponenta interpretu Pythonu, která překládá zdrojový kód Pythonu do bytecode. Tento bytecode je pak interpretován Python Virtual Machine (PVM) k provedení operací specifikovaných v kódu Pythonu. Kompilátor Pythonu je zodpovědný za generování bytecode z lidsky čitelného zdrojového kódu Pythonu.

- Překládá celý zdrojový kód do strojového kódu před spuštěním.
- Detekuje chyby v kódu během fáze kompilace.

Interpret

Python interpreter je program, který čte a vykonává kód napsaný v jazyce Python. Překládá Python kód do strojově čitelného bytecode, který je poté spuštěn pomocí Python Virtual Machine (PVM). Interpret Pythonu kontroluje syntaxi kódu, interpretuje ho řádek po řádku a provádí instrukce specifikované v kódu. Python interpret umožňuje uživatelům interaktivně spouštět Python kód, což z něj činí klíčovou součást při provádění Python programů.

- Překládá zdrojový kód řádek po řádku za běhu a kontroluje syntaxi.
- Vykonává kód přímo bez generování mezilehlého spustitelného souboru.
- Detekuje chyby za běhu při interpretaci každého řádku.

Python - převod zdrojového kódu na spustitelný kód

1. Zdrojový kód je vytvořen v editoru a následně uložen (.py soubor).
2. Kompilace zdrojového souboru Python kompiluje zdrojový kód do *byte code*, kompilátor též kontroluje syntaxi a vytváří .pyc soubor.
3. Byte code Kód v *byte code (.pyc)* je předán do Python Virtual Machine[1](PVM) což je Python interpreter. PVM konvertuje *byte code* do *machine-executable code* (binární jazyk složený z 0 a 1 kterému rozumí CPU systému)a čte a vykonává soubor řádku po řádce.
4. Spuštění programu CPU spustí machine code.

[1]Virtuální stroj byl v Pythonu použit z důvodů přenositelnosti

Různé implementace Pythonu

Python je jazyk s určitými pravidly syntaxe a sémantiky, ale implementace Pythonu může být provedena různě. Výchozí a nejčastěji používaná implementace Pythonu je CPython.

- **CPython**: Standardní a nejpoužívanější implementace Pythonu napsaná v jazyce C.
- **Jython**: Implementace Pythonu, která běží na platformě Java Virtual Machine (JVM). Umí kombinovat Python s existujícím Java kódem.
- **IronPython**: Implementace Pythonu pro platformu .NET Framework. Umožňuje integraci Pythonu s .NET knihovnami a kódem.
- **PyPy**: Implementace Pythonu s důrazem na rychlost. Používá JIT (Just-In-Time) kompilaci k optimalizaci výkonu.
- **MicroPython**: Odlehčená implementace Pythonu optimalizovaná pro vestavěné systémy a mikrokontroléry.

První aplikace v jazyce Python

První aplikace v jazyce Python

```
1  # Vytvarime prvni aplikaci v jazyce Python
2
3  5
4  3.4
5  cislo1 = 10
6  cislo2 = 12
7  cislo4 = cislo1 * cislo2;
8  cislo4 % 3
9  print(cislo1 + cislo2) # vytiskne soucet cislo1 a cislo2
10
11
12 cislo3 = (25 + 35)**2
13 print(cislo3)
14 print (24 / 7)
15 print (24 // 5)
```

Přednáška 2

Základní datové typy, reprezentace čísel v počítači

Co je datový typ v jazyce Python?

- Datový typ v jazyce Python definuje typ dat, která může proměnná[2] obsahovat.
- Určuje operace, které lze provádět s daty.
- Python je dynamicky typovaný jazyk, což znamená, že není nutné explicitně deklarovat datový typ.
- Python automaticky přiřazuje datový typ na základě hodnoty přiřazené proměnné.

[2] Uvědomuji si, že pojem proměnnou budeme probírat později.

Vestavěné datové typy v jazyce Python

int - **celá čísla** bez desetinné čárky

float - **desetinná čísla** s plovoucí desetinnou čárkou nebo ve tvaru s exponentem

complex - **komplexní čísla** s reálnou a imaginární částí

str - **řetězce** - posloupnosti znaků uzavřené v uvozovkách

list - **seznamy** položek, které lze upravovat a jsou uspořádané

tuple - **n-tice** - spořádané kolekce položek, které nelze měnit

dict - **slovníky** - páry klíč-hodnota, které umožňují rychlé vyhledávání

set - **množiny** - neuspořádané kolekce jedinečných prvků

bool - **booleovská hodnota** reprezentuje pravdivostní hodnoty, buď True nebo False

NoneType - **žádná hodnota** reprezentuje absenci hodnoty nebo null

Vestavěné číselné datové typy v jazyce Python

- Python má několik vestavěných číselných datových typů pro reprezentaci různých druhů číselných hodnot[3]:

  int - **celá čísla** bez desetinné čárky.

  float - **desetinná čísla** s plovoucí desetinnou čárkou nebo ve tvaru s exponentem.

  complex - **komplexní čísla** s reálnou a imaginární částí.

[3] Datový typ `bool` je podmnožinou datového typu `int`, bude probírán dále

Počítače a binární čísla

Počítače a binární čísla

- Počítače používají binární čísla, která jsou složena z nul a jedniček, k reprezentaci dat a provádění výpočtů - tzn. veškerá digitální data jsou nakonec reprezentována v binární formě
- Binární čísla jsou čísla se základem 2, používající pouze 0 a 1 jako číslice.
- Každá číslice v binárním čísle se nazývá bit (binární číslice).
- Binární čísla se používají v počítačových systémech pro ukládání paměti, aritmetické operace a logické operace.

Ukázka reprezentace čísla 47 binárním číslem (předpoklad, že používáme pro číslo 4 byty):

$$(00000000 \quad 00000000 \quad 00000000 \quad 00101111)_2 = (47)_{10}$$

$$1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 = 47$$

Reprezentace záporných celých čísel v počítačích

- V počítačích se záporná celá čísla reprezentují pomocí systému nazvaného *doplňkový kód (Two's Complement)*.
- Doplňkový kód umožňuje jednoduchou reprezentaci a manipulaci s kladnými a zápornými čísly ve stejném systému.
- V Doplňkovém kódu se nejlevější bit používá jako znaménkový bit[4]. Pokud je tento bit 0, číslo je kladné; pokud je 1, číslo je záporné.
- Příklad pro 8 bitové číslo (-70) a 16 bitové číslo (-24761):

-128	64	32	16	8	4	2	1
1	0	1	1	1	0	1	0

-32768	16384	8192	4096	2048	1024	512	256
1	0	0	1	0	0	1	0

128	64	32	16	8	4	2	1
1	0	0	1	0	0	1	0

[4] Existuje i jiný způsob.

Velikost a rozsah

- Pro reprezentaci magnitudy záporného čísla se inverzí všech bitů kladné hodnoty a přidá se 1 k výsledku.
- Příklad: Pro reprezentaci čísla -5 se invertními bity hodnoty 5 (00000101) a přičtením 1 získáme Two's Complement reprezentaci -5 jako 11111011.
- V 8-bitovém systému je rozsah reprezentovatelných hodnot pomocí Doplňkového kódu od -128 do 127. Nejlevější bit (znaménkový bit) určuje, zda je číslo kladné nebo záporné.

Počet bitů	Vzorec	Rozsah
8	-2^{8-1} to $2^{8-1}-1$	-128 do 127
16	-2^{16-1} to $2^{16-1}-1$	$-32,768$ do $32,767$
24	-2^{24-1} to $2^{24-1}-1$	$-8,388,608$ do $8,388,607$
32	-2^{32-1} to $2^{32-1}-1$	$-2,147,483,648$ do $2,147,483,647$

Převod binárního na desítkový systém

- Převod binárních čísel na desítkový systém zahrnuje násobení každého bitu 2 umocněného na jeho pozici.
- Sčítáním výsledků získáte desetinný ekvivalent binárního čísla.
- Například binární číslo 11010 je ekvivalentní desetinnému číslu 26.
  Ukázka reprezentace čísla 47 a 235061 binárním číslem (předpoklad, že používáme pro číslo 4 byty):

$$(00000000 \quad 00000000 \quad 00000000 \quad 00101111)_2 = (47)_{10}$$

$$1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 + 1 \times 2^0 = 47$$

$$(00000000 \quad 00000011 \quad 10010110 \quad 00110101)_2 = (235061)_{10}$$

$$2^{17} + 2^{16} + 2^{15} + 2^{12} + 2^{10} + 2^9 + 2^5 + 2^4 + 2^2 + 2^0 = 235061$$

Převod z desítkového na binární systém

1. Inicializujme prázdný seznam pro ukládání binárních číslic.
2. Pro dané desítkové číslo `číslo` spustíme smyčku, dokud `číslo` je větší než 0.
3. Uvnitř smyčky vypočítáme zbytek po dělení `číslo` číslem 2 (toto je nejméně významná číslice).
4. Přidáme vypočítaný zbytek na začátek seznamu binárních číslic.
5. Číslo `číslo` celočíselně vydělíme číslem 2 a aktualizujeme `číslo` s celočíselným výsledkem dělení.
6. Opakujeme kroky 3-5, dokud `číslo` není 0.
7. Binární reprezentace desítkového čísla je spojení binárních číslic ze seznamu.

Převod z desítkového na binární systém - příklad a kroky algoritmu

- Převod desítkového čísla 177 na binární.
- Tento symbol značí celočíselné dělení $\div$

177	$\div$	2	=	88	1
88	$\div$	2	=	44	0
44	$\div$	2	=	22	0
22	$\div$	2	=	11	0
11	$\div$	2	=	5	1
5	$\div$	2	=	2	1
2	$\div$	2	=	1	0
1	$\div$	2	=	0	1

- Výsledné binární číslo je 10110001

Binární aritmetika

- Binární aritmetika zahrnuje sčítání, odčítání, násobení a dělení binárních čísel.
- **Binární Sčítání** je podobné desítkovému sčítání, s rozdílem, že přenos nastává při dosažení hodnoty 2 místo 10. Sčítání v binární soustavě funguje podle následující tabulky:

$0 + 0$	$=$	0
$0 + 1$	$=$	1
$1 + 0$	$=$	1
$1 + 1$	$=$	10 (přenos 1)

- **Binární odčítání** je podobné desítkovému odčítání, s použitím půjčky, když je potřeba. Odčítání v binární soustavě funguje podle následující tabulky:

0 - 0	$=$	0
1 - 0	$=$	1
1 - 1	$=$	1
0 - 1	$=$	10 (půjčka 1)

Reprezentace reálných čísel v počítači - aritmetika v plovoucí řádové čárce

- **Úloha** 1: Pomocí počítače vynásobme dvě reálná čísla $\frac{1}{10} \times \frac{2}{5}$ a porovnejme výsledek s hodnotou $\frac{2}{50}$. Tato úloha vypadá na první pohled naprosto nesmyslně (výsledek známe), přesto se ji pokusme nyní věnovat.

- **Úloha** 2: Pomocí počítače vyřešme následující soustavu rovnic (2×3):

$$\begin{aligned} (10^{14}+7)x + (10^{14}+1)y &= 2(10^{14}+4) \\ (10^{14}+1)x + (10^{14}+1)y &= 2(10^{14}+1) \end{aligned} \tag{1}$$

respektive (přepsáno méně čitelnou formou):

$$\begin{aligned} 100000000000007x + 100000000000001y &= 200000000000008 \\ 100000000000001x + 100000000000001y &= 200000000000002 \end{aligned}$$

Řešení úlohy 2

Řešení této soustavyje triviální: Od horního řádku odečteme řádek spodní a ihned získáme hodnotu x

$$6x + 0y = 6 \rightsquigarrow x = 1$$

dosazením vypočteného x do řádku dvě vypočteme hodnotu neznámé y

$$100000000000001 + 100000000000001y = 200000000000002 \rightsquigarrow y = 1$$

Nechme soustavu rovnic (1) vyřešit počítačem, resp. programem který dokáže řešit soustavy rovnic `GNU Octave`.

```
octave:1>  K = 10^14; K =  100000000000000
octave:2> A = [K + 7, K + 1;
>              K + 1, K + 1]
A =

   100000000000007   100000000000001
   100000000000001   100000000000001
```

Řešení úlohy 2 ...

```
octave:3> L = [2*K + 8;
>              2*K + 2]
L =
   200000000000008
   200000000000002

octave:4> A \ L
ans =
   1.001599629720052
   0.998400370279948
```

Výsledek je však chybný! Výsledek je zapříčiněn vlastnostmi čísel zobrazených v pohyblivé řádové čárce, resp. realizací reálných čísel na počítači.

Realizace a zobrazení čísel v pohyblivé řádové čárce

- Počítače jsou diskrétní, konečné stroje a proto reálná čísla v počítačích nemohou být uložena s nekonečnou přesností — reálná čísla v počítačích neexistují.
- Reálná čísla v počítači jsou interně zobrazena (realizována) pomocí tří částí:
  - **mantisa** obsahuje číslo v tzv. normalizovaném tvaru
  - **exponent** určuje, o kolik řádů je nutno posunout řádovou čárku.
  - **znaménkový bit** určuje znaménko
- Počet bitů mantisy a exponentu je implementačně závislý a liší se dle typu počítače a zvoleného reálného datového formátu.
- Standard IEEE 954 definuje čtyři (zde mi chybí single extended) formáty zobrazení čísel v pohyblivé řádové čárce:

Formát	Délka mantisy	Celková délka formátu
`single`	24	32
`double`	53	64
`double extended`	$\geqslant 64$	$\geqslant 79$

Tabulka: Velikosti datových formátů (počet bitů)

Realizace a zobrazení čísel v pohyblivé řádové čárce

- Počet bitù určených pro reprezentaci reálného čísla uloženého v paměti počítače je vždy **omezen**
- Není možno korektně zobrazit všechna čísla na číselné ose
- Čím větší je počet bitù v kterých je číslo uloženo, tím je formát ‘přesnější’ a má větší “rozsah”
- Počet všech čísel, které je možno pomocí daného formátu zobrazit je hodnota, kterou dává k dispozici kódování do n bitů — tedy počet reálných čísel zakódovaných v n bitech může maximálně být 2^n.

Konkrétní reprezentace reálných čísel

Reálná čísla v počítači jsou interně zobrazena (realizována) pomocí tří částí, *znaménkového bitu*, *mantisy* a *exponentu*.
Označme znaménkový bit s, mantisu m a exponent e. Poté číslo realizováno pomocí s, m a e má tvar

$$-1^s \times M \times B^{e-E},$$

kde B je báze soustavy reprezentace (téměř vždy 2) a E je "bias" – celočíselná konstanta daná typem implementace.
Pokud počet bitů mantisy označíme r_m, mantisa má následující tvar

$$d_0 d_1 \ldots d_{r_m-1}$$

a reprezentuje reálné èíslo

$$-1^s \times \left(1 + d_0 B^{-1} + d_1 B^{-2} + \cdots + d_{r_m-1} B^{-r_m}\right) \times B^e, \ (0 \leqslant d_i < B). \quad (2)$$

Konkrétní reprezentace reálných čísel - přesnost a rozsah

Rozsah mantisy určuje relativní přesnost zobrazení čísla, rozsah čísel lze zjistit z velikosti exponentu:

Přesnost Dvě sousední čísla se liší v hodnotě nejnižšího bitu mantisy (zatím nechme stranou velikost e a skutečnost, že se sousední čísla liší o hodnotu nejnižšího bitu mantisy $\times B^{e-E}$). Pokud je rozsah mantisy např. 22, liší se dvě bezprostředně po sobě následující čísla o hodnotu 2^{-22}, tedy o 2.4×10^{-7}. Čísla tedy lze používat s přesností na 7 platných číslic.

Rozsah Pokud rozsah (počet bitù) exponentu označíme r_e získáme v exponentu rozmezí $[-E, 2^{r_e} - E]$. Například pro $r_e = 8$ je posun desetinné tečky v rozsahu $\left[2^{-127}, 2^{128}\right]$ a čísla se pohybují v intervalu 2^{-127} a$\frac{3}{4}$ 2^{128}, přibližně tedy v intervalu 10^{-38} až 10^{+38}.

Konkrétní reprezentace reálných čísel - přesnost a rozsah

Jedním z důsledků rovnice (2) je existence nekonečného počtu reálných čísel majících v desítkové soustavě konečné vyjádření oproti nekonečnému vyjádření v soustavě dvojkové. Například číslo $\frac{1}{10}$ je ve dvojkové soustavì vyjádřitelné pouze nekonečným binárním rozvojem.

- Ukázky čísel které jsou v desítkové soustavě vyjádřené konečným číslem, ve dvojkové soustavě nekonečným binárním rozvojem:

desítková soustava	**dvojková soustava**
0.1	$0.0001100110011\ldots$
0.2	$0.001100110011\ldots$
0.3	$0.01001100110011\ldots$
0.4	$0.011001100110011\ldots$
0.6	$0.1001100110011\ldots$

Konkrétní reprezentace reálných čísel - přesnost a rozsah

■ **Příklad:** Převeďme číslo 34.323 do normalizovaného tvaru pohyblivé řádové čárky. Rozsah mantisy + znaménkový bit je $r_m = 24$ bitù, rozsah exponentu je $r_e = 8$ a bias $E = 127$.

Nejprve převedeme celou část čísla 34.323: $(34)_{10} = (100010)_2$. Pro kontrolu

$$1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 32 + 2 = 34$$

Desetinnou část dělíme postupně číslem 2 a sledujeme, zdali je výsledek větší než jedna, pokud ano zapíšeme binární jedničku a pro další násobení jedničku odečteme, v opačném případě zapíšeme binární 0. Tento cyklus opakujeme tak dlouho, dokud výsledek není nulový. Desetinná část, číslo 0.323, nemá konečné binární vyjádření

$$(0.323)_{10} = (0.01010010101100000010000011\ldots)_2$$

Konkrétní reprezentace reálných čísel - přesnost a rozsah

kontrola

$$0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + \ldots 1 \times 2^{-26} = 0.322999998927116$$

Binární reprezentace čísla 34.323

$$(34.323)_{10} \approx (100010.01010010101100000010000011000100100110111101)_2$$

Jelikož číslo je kladné, znaménkový bit je roven 0. Při výpočtu čísla dle rovnice se (2) musíme posunout desetinnou tečku tak, abychom dostali tvar $1.m \times 2^{posun}$ a tedy posunout desetinnou tečku v kladném směru o pět pozic.
Číslo 34.323 je reprezentováno naším formátem následovně (znak | naznačuje začátek mantisy a exponentu):

$$0|00010010100101011000000|10000100$$

Číslo 34.323 nelze ve dvojkové soustavì zobrazit přesně:

$$(34.323)_{10} \approx (100010.0101001010110000)_2 = (34.322998)_{10}$$

Reprezentace čísel ve formátu `single`

Formát `single` se skládá ze tří polí: 23-bitové mantisy m, 8bitového exponentu e a znaménkového bitu s. Tato pole jsou souvisle uložena v 32 bitech:

s	e[30:23]	m[22:0]
31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Ostatní IEEE formáty (`double`, `extended single` a `extended double`) mají velmi podobnou representaci — hlavní rozdíl je v počtech bitù vyčleněných pro mantisu a exponent a ve velikosti biasu.

IEEE `single` formát	**Bitù mantisy**	**Min. kladné èíslo**	**Max. kladné èíslo**	**Poèet platných cifer**
`single`	24	1.175×10^{-38}	$3.402 \times 10^{+38}$	6–9
`double`	53	2.225×10^{-308}	$1.797 \times 10^{+308}$	15–17
`double extended`	64	3.362×10^{-4932}	$1.189 \times 10^{+4932}$	18–21

Proměnné v jazyce Python

Proměnné v jazyce Python

- Proměnná je označení pro místo v paměti, kde je uložena hodnota.
- Jazyk Python nemá žádný příkaz pro deklaraci proměnných, proměnnou vytvoříte v okamžiku kdy ji poprvé přiřadíte hodnotu
- Proměnné může být přiřazena hodnota daného typu a později může být přiřazena hodnota jiného typu
- Název proměnné by měl být výstižný a popisný, aby bylo jasné, co proměnná reprezentuje - požadavky na jmennou konvenci:
  - Jméno proměnné by měla začínat malým písmenem nebo podtržítkem _
  - Jméno proměnné nesmí začínat číslem
  - Jméno proměnné může obsahovat pouze alpha-numeric znaky a podtržítko (A-z, 0-9, a _)
  - Jména proměnných jsou citlivá na velké a malé písmena
  - Jméno proměnné nemůže být žádné rezervované slovo jazyka Python
- Přiřazení hodnoty proměnné se provádí pomocí operátoru =
- Python je dynamicky typovaný jazyk, což znamená, že typ proměnné se určuje automaticky podle hodnoty, která je do proměnné přiřazena

Proměnné v jazyce Pythonu

```
>>> cislo = 10
>>> cislo
10
>>> cislo1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cislo1' is not defined. Did you mean: 'cislo'?
>>>
>>> soucetLet = 1934
>>> dvojnasobekSouctuLet = 2*soucetLet
>>>
>>> dvojnasobekSouctuLet
3868
>>>
>>> cislo = "Dnes je krasne"
>>>
>>> cislo
'Dnes je krasne'
>>>
```

Příkazy type and print

- Příkaz `type` slouží k zjištění typu objektu v Pythonu.
- Syntaxe: `type(object)`
- Příklad: `x = 10, print(type(x))`
- Výstup: `<class 'int'>`‘

- Příkaz `print` slouží k výpisu textu a hodnot na standardní výstup.
- Syntaxe: `print(value1, value2, ...)`
- Příklad: `x = 10, print("Hodnota x je:", x)`
- Výstup: `Hodnota x je: 10`

Celá čísla

Celá čísla (int)

- Celá čísla v Pythonu reprezentují celá čísla bez desetinných míst
- Mohou být kladná, záporná nebo nula
- Celá čísla v Pythonu mají neomezenou přesnost, což znamená, že mohou být libovolně velká nebo malá (pozor na rychlost výpočtu u velkých čísel)
- Operace s celými čísly zahrnují sčítání, odčítání, násobení a dělení
- Příklady: 1, 2, 3, 4, ...

Proměnné v jazyce Pythonu

```
>>> 5 + 30 # scitani
35
>>> 45 - 20 # rozdil
25
>>> 6 / 4 # deleni
1.5
>>> 10 / 5 # deleni
2.0
>>> 2 ** 4 # umocnovani
16
>>> (10 + 43) * 3
159
>>> _ + 10
169
>>> 20 % 7 # zbytek po deleni
6
>>> 10 // 4 # celociselne deleni
2
>>>
```

Operátory jazyka Python (dle priority — od největší k nejmenší)

Operace	Popis
()	závorky
**	exponent
$+x$, $-x$, $\sim x$	unární +, unární −, bitové NOT
*, /, //, %	násobení, dělení, celočíselné dělení, zbytek po dělení
+, −	sčítání, odčítání
<<, >>	bitové posuny
&	bitové AND
	bitové XOR
|	bitové OR
==, ! =, >, >=, <, <=, is, is not, in, not in	porovnání, operátory členství
not	logické NOT
and	logické AND
or	logické OR

Použití operátorů

```
>>> 5 + 10 * 20 ** 2
4005
>>> (5 + 10) * 20 ** 2
6000
>>> (5 + 10 * 20) ** 2
42025
>>> 10 + (20 // 7) * 5
20
```

Desetinná čísla

Desetinná čísla (float)

- Desetinná čísla v Pythonu reprezentují čísla s desetinnými místy nebo ve formě exponentu
- Používají se k reprezentaci reálných čísel a mohou mít desetinnou část
- Python používá standard IEEE 754 k reprezentaci desetinných čísel
- Operace s desetinnými čísly mohou někdy vést k chybám přesnosti, viz reprezentace reálných čísel v počítači

Desetinná čísla (float)

```
>>> cislo = 1.453345
>>> cislo * 34.4
49.995068
>>> cislo1 = 12.3e6
>>> cislo1
12300000.0
>>>
```

Module `math`

Konstanty a Funkce v Modulu Math

- Modul `math` v Pythonu poskytuje matematické funkce a konstanty
- Pro použití modulu je potřeba ho nejprve importovat: `import math`

Konstanta/Funkce	**Popis**
`math.pi`	Matematická konstanta π
`math.e`	Matematická konstanta Eulerovo číslo e
`math.sqrt(x)`	Odmocnina čísla x
`math.sin(x)`	Sinus úhlu x v radiánech
`math.cos(x)`	Kosinus úhlu x v radiánech
`math.exp(x)`	Exponenciální funkce e^x

Komplexní čísla

Komplexní čísla (complex)

- Komplexní čísla v Pythonu mají reálnou a imaginární část
- Jsou reprezentována jako $a + bj$, kde a je reálná část a b je imaginární část.
- Python poskytuje vestavěné funkce pro práci s komplexními čísly, jako jsou `complex()`, `real` a `imag`
- Komplexní čísla se používají v různých matematických a vědeckých výpočtech
- Příklad: z = 2 + 3j

Komplexní čísla (complex)

```
>>> cislo = 5 + 6j
>>> cislo
(5+6j)
>>> cislo1 = 4 - 9j
>>> cislo1
(4-9j)
>>> cislo + cislo1
(9-3j)
>>> cislo ** 2
(-11+60j)
>>> cislo.real
5.0
>>> cislo.imag
6.0
>>> cislo2 = complex(1,2)
>>> cislo2
(1+2j)
>>> cislo.imag
6.0
```

Datový typ Bool

Datový typ Bool

- Reprezentuje logické hodnoty - `True` (pravda) nebo `False` (nepravda)
- `True` představuje logickou hodnotu pravda
- `False` představuje logickou hodnotu nepravda
- Používá se v podmínkách a logických operacích
- Produkují logické výsledky (např. $==$, $!=$, $>$, $<$).
- Používají se k kombinaci logických výrazů (např. `and`, `or`, `not`):

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Datový typ Bool v jazyce Python

```
>>> 1 == 1
True
>>> 1 != 1
False
>>> 1 != 1 or 2 == 2
True
>>> 1 != 1 or 2 == 2
True
>>> 2 < 90
True
>>> 234 < 40
False
>>> 234 < 40 and 4 > 103
False
>>> 234 < 40 or 4 > 103
False
>>>
```

Řetězce

Datový typ Řetězec (String) v Pythonu

- Reprezentuje textové hodnoty v Pythonu
- Řetězec je sekvence znaků uzavřená v jednoduchých ' nebo dvojitých "uvozovkách.
- Python poskytuje mnoho funkcí pro práci s řetězci, jako je spojování, rozdělování, porovnávání atd.
- Řetězce lze indexovat a řezat pro získání konkrétních částí
- Je reprezentován jako pole znaků, každý znak v řetězci má svůj index (počínaje od 0)
- Délku řetězce vrací funkce `len(řetězec)`
- Ukázky indexace:
  - Řetězec "Ahoj"má indexy: A(0), h(1), o(2), j(3)
  - Přístup k 8. znaku: "Ahoj, dnes je úterý"[8]

Řetězce - ukázka

```
>>> retezec1 = "Dnes je krasne"
>>> retezec2 = 'Ahoj'
>>> print(retezec1)
Dnes je krasne
>>> print(retezec2)
Ahoj
>>> type(retezec1)
<class 'str'>
>>> retezec3 = retezec1 + retezec2 # spojovani retezcu
>>> print(retezec3)
Dnes je krasneAhoj
>>> print(retezec3[0]) # indexovani retezcu - prvni znak v
    retezci
D
>>> print(retezec3[1]) # indexovani retezcu - druhy znak v
    retezci
n
>>> print(retezec3[100]) # pristup k neexistujicimu znaku
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Nejpoužívanější funkce pro práci s řetězci v Pythonu

Název funkce	Stručné vysvětlení
`len()`	Vrací délku řetězce
`str.lower()`	Převede všechny znaky v řetězci na malá písmena
`str.upper()`	Převede všechny znaky v řetězci na velká písmena
`str.strip()`	Odstraní všechny vedoucí (na začátku) a koncové (na konci) mezery
`str.split(oddělovač)`	Rozdělí řetězec na seznam podle zadaného oddělovače
`str.join(iterable)`	Spojuje prvky iterovatelného objektu (např. seznam) do řetězce s určeným oddělovačem
`str.replace(pod, str)`	Nahradí zadaný podřetězec jiným podřetězcem v řetězci

Řetězce - nejpoužívanější funkce

```
>>> # Delka retezce
>>> retezec1 = "Jazyk Python je skvely"
>>> length = len(retezec1)
>>> print("Delka retezce:", length)
Delka retezce: 22
>>> # Rozdelovani retezcu
>>> text = "Hello, World!"
>>> substring = text[7:]
>>> print("Podretezec:", substring)
Podretezec: World!
>>> # Prevod na male a velke znaky
>>> sentence = "Programovani v Pythonu je zabava"
>>> uppercase_sentence = sentence.upper()
>>> lowercase_sentence = sentence.lower()
>>> word_list = sentence.split()
>>> print("Uppercase:", uppercase_sentence)
Uppercase: PROGRAMOVANI V PYTHONU JE ZABAVA
>>> print("Lowercase:", lowercase_sentence)
Lowercase: programovani v pythonu je zabava
```

Řetězce - nejpoužívanější funkce

```
>>> print("Word List:", word_list)
Word List: ['Programovani', 'v', 'Pythonu', 'je', 'zabava']
>>> # Nahrazovani textu
>>> quote = "Life is short, use Python!"
>>> new_quote = quote.replace("Python", "Java")
>>> print("New Quote:", new_quote)
New Quote: Life is short, use Java!
>>> # Hledani v textu
>>> email = "jan.novak@gmail.com"
>>> if "@" in email:
...     print("Spravna adresa")
... else:
...     print("nespravna adresa")
...
Spravna adresa
```

Escape characters

Speciální symboly uvozené lomítkem:

Speciální znak	Význam
\n	Nový řádek
\t	Odstup
\'	Jednoduchá uvozovka
\"	Dvojitá uvozovka
\\	Zpětné lomítko

Unicode v Pythonu

- Unicode je standard pro reprezentaci znaků v různých jazycích a písmech.
- V Pythonu lze pracovat s Unicode znaky pomocí escape sekvencí.

Unicode	Popis
U+0161	Malé písmeno š
U+0158	Velké písmeno Ř
U+0041	Velké písmeno A
U+00E1	Malé písmeno á
U+03B1	Řecké písmeno α

Přednáška 3

Odvozené datové typy

Seznamy v Pythonu

Seznamy v Pythonu

- Seznamy jsou uspořádané, měnitelné a všestranné datové struktury, které mohou uchovávat kolekci prvků.
- Jsou vytvořeny umístěním prvků do hranatých závorek []
- Seznamy slouží k ukládání více prvků do jedné proměnné
- Mohou obsahovat prvky různých datových typů
- Seznamy jsou indexovatelné
- Podporují různé operace, jako je přidávání, odebírání, řazení a další
- Nabízejí širokou škálu funkcí pro manipulaci a přístup

Funkce seznamů

Funkce	Syntaxe	Popis
Append	`list.append(element)`	Přidá prvek `element` na konec seznamu.
Insert	`list.insert(index, element)`	Vloží prvek `element` na určenou pozici `index` v seznamu.
Remove	`list.remove(element)`	Odebere první výskyt zadané hodnoty `element` ze seznamu.
Pop	`popped_element = list.pop(index)`	Odebere a vrátí prvek na určeném indexu `index` ze seznamu.
Sort	`list.sort()`	Seřadí seznam vzestupně.
Index	`index = list.index(element)`	Vrátí index prvního výskytu zadané hodnoty `element` v seznamu.
Len	`length = len(list)`	Vrátí počet prvků v seznamu.

Ukázky použití

```
# Priklad 1: Pridani
my_list = [1, 2, 3]
my_list.append(4)
# Priklad 2: Vlozeni
my_list.insert(2, 2.5)
# Example 3: Odebrani
my_list.remove(2)
# Example 4: Odebrani prvku
popped_element = my_list.pop(2)
# Example 5: Serazeni
my_list.sort()
# Example 6: Index
index = my_list.index(3)
# Example 7: Pocet prvku
length = len(my_list)
# Example 8: Count
count = my_list.count(2)
# Example 9: Rozsireni
new_list = [5, 6, 7]
my_list.extend(new_list)
```

N-tice v Pythonu

N-tice v Pythonu

- N-tice v Pythonu jsou nezměnitelné datové struktury, které mohou uchovávat kolekci prvků různých typů. Jsou vytvořeny pomocí závorek ().
- N-tice jsou podobné seznamům, ale nejsou měnitelné.
- Mohou obsahovat prvky různých datových typů.
- N-tice jsou indexovatelné a lze přistupovat k prvkům podobně jako u seznamů.
- N-tice mají omezené funkce ve srovnání se seznamy, protože jsou nezměnitelné.
- Nicméně, podporují základní operace jako indexování a slicing.

Nejčastěji používané funkce nad N-ticemi

Function	Syntax	Description
Index	`index = tuple.index(element)`	Vrátí index prvního výskytu za dané hodnoty `element` v tuplu
Count	`count = tuple.count(element)`	Vrátí počet výskytů zadan hodnoty `element` v tuplu.
Length	`length = len(tuple)`	Vrátí počet prvků v tuplu.

N-tice - ukázka

```
>>> # Vytvoreni N-tic
>>> n1 = (1, 2, 'jablko', 'banan')
>>> # Vypsani prvniho a posledniho elementu
>>> print("Prvni element:", n1[0])
Prvni element: 1
>>> print("Posledni element:", n1[-1])
Posledni element: banan
>>> # Delka N-tice
>>> print("Delka n-tice", len(n1))
Delka n-tice 4
>>> # Pocitani vyskytu
>>> count = n1.count("ahoj")
>>> print("Pocet vyskytu ahoj:", count)
Pocet vyskytu ahoj: 0
>>> # Index funkce
>>> index = n1.index('banan')
>>> print("Index 'banan':", index)
Index 'banan': 3
>>> #
>>> a, b, ovoce1, ovoce2 = n1
>>> print("Hodnoty:", a, b, ovoce1, ovoce2)
```

Slovníky v Pythonu

Slovníky v Pythonu

- Slovníky v Pythonu jsou neuspořádané kolekce datových struktur, které uchovávají páry klíč-hodnota. Jsou vytvořeny pomocí složených závorek {}.
- Slovníky umožňují rychlý přístup k hodnotám pomocí klíčů.
- Každý klíč v slovníku musí být unikátní.
- Hodnoty v slovníku mohou být různých datových typů.
- Slovníky v Pythonu jsou užitečné pro uchovávání dat v podobě klíč-hodnota

Funkce slovníků

Funkce	Syntax	Příkla
Přístup k Hodnotě	`hodnota = slovník[klíč]`	`věk =`
Přidání/Aktualizace Hodnoty	`slovník[klíč] = hodnota`	`osoba York'`
Odstranění Klíč-Hodnota Páru	`del slovník[klíč]`	`del o`
Kontrola Existence Klíče	`if klíč in slovník:`	`if 'm`
Seznam Klíčů	`klíče = list(slovník.keys())`	`klíče list(`
Seznam Hodnot	`hodnoty = list(slovník.values())`	`hodno list(`
Seznam Položek	`položky = list(slovník.items())`	`polož list(`

Tabulka: Nejpoužívanější Funkce pro Slovníky v Pythonu

Slovníky - ukázka

```
>>> # Vytvoreni slovniku
>>> subject = {'jmeno': 'Uvod do jazyka Python', 'kreditu':
    5, 'zakonceni': 'zkouska'}
>>> # Ziskani jmeno predmetu Value
>>> jmeno = subject['jmeno']
>>> print("Jmeno predmetu:", jmeno)
Jmeno predmetu: Uvod do jazyka Python
>>> # Pridani/Modifikace hodnot
>>> subject['kreditu'] = 5
>>> print("Nove kredity:", subject['kreditu'])
Nove kredity: 5
>>> # Smazani polozky s danym klicem
>>> del subject['zakonceni']
>>> print("Dictionary after removing 'city':", subject)
Dictionary after removing 'city': {'jmeno': 'Uvod do jazyka
    Python', 'kreditu': 5}
>>> # Kontrola existence klice
>>> if 'kreditu' in subject:
...     print("Klic kreditu exituje ve slovniku existuje")
```

Slovníky - ukázka

```
>>> # List klicu
>>> klice = list(subject.keys())
>>> print("Klice:", klice)
Klice: ['jmeno', 'kreditu']
>>> # List hodnot
>>> hodnoty = list(subject.values())
>>> print("Hodnoty:", hodnoty)
Hodnoty: ['Uvod do jazyka Python', 5]
>>> # Items list
>>> items = list(subject.items())
>>> print("Items:", items)
Items: [('jmeno', 'Uvod do jazyka Python'), ('kreditu', 5)]
```

Přednáška 4

Základní booleovské funkce, podmíněný příkaz

Podmínky

Podmínky If, Elif, Else v Pythonu

- Python používá příkazy `if`, `elif` a `else` pro podmíněné provádění kódu.
- Příkaz `elif` umožňuje definovat další podmínky po `if`.
- Syntaxe:

```
if podmínka1:
    kód
elif podmínka2:
    kód
```

- Příklad:

```
x = 10
if x > 5:
    print("x je větší než 5")
```

If - syntaxe

```
1 cislo = 10
2
3 if cislo > 0:
4     print("Cislo je kladne")
5 elif cislo < 0:
6     print("Cislo je zaporne")
7 else:
8     print("Cislo je nula")
```

Cyklus for

For cyklus v Pythonu

- For cyklus v Pythonu slouží k iteraci přes sekvenci (např. list, tuple, string) nebo rozsah čísel.
- Funkce range() je běžně používána k vytvoření sekvence čísel.
- Příkaz break slouží k předčasnému ukončení smyčky.
- Příkaz continue slouží k přeskočení zbytku aktuální iterace a pokračování na další.

Python For Loop

- Cyklus for v jazyce Python slouží k opakování určitého bloku kódu pro každý prvek v sekvenci (jako je list, tuple, dictionary, string atd.).
- item Syntaxe:

```
1 for prvek in sekvence:
2     blok kodu k opakovani
```

- Proměnná **prvek** zastupuje aktuální prvek v sekvenci.
- **Sekvence** může být jakýkoli iterovatelný objekt.

Funkce range() v Pythonu

- Funkce range() v Pythonu generuje sekvenci čísel.
- Syntaxe: `range(start, stop, step)`
- Parametry:
  - `start`: Počáteční hodnota (výchozí hodnota je 0).
  - `stop`: Koncová hodnota (nebude zahrnuta).
  - `step`: Krok (výchozí hodnota je 1).
- Příklady:
  - `range(5)` generuje čísla od 0 do 4.
  - `range(1, 6, 2)` generuje lichá čísla od 1 do 5.

Funkce range() v Pythonu — ukázky použití

```
>>> # funkce range nevraci list, pouze to tak "vypada"
>>> print(range(10))
range(0, 10)
>>>
>>> print (list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> print (list(range(0,20,2)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>
>>> print (list(range(100,0, -5)))
[100, 95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35,
    30, 25, 20, 15, 10, 5]
```

For cyklus - různé průchody

- Použití range() ve for cyklu

```
for i in range(5):
    print(i)
```

- Použití break ve for cyklu

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

- Použití continue ve for cyklu

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

For - různé průchody

- Iterace přes List

```
1 ovoce = ['jablko', 'banan', 'tresen']
2 for ovoce in ovoce:
3     print(ovoce)
```

- Iterace přes String

```
1 slovo = 'Python'
2 for pismeno in slovo:
3     print(pismeno)
```

- Iterace přes Slovník

```
1 studenti = {1234 : 'Jan Novak', 3456: 'Jana Nova', 345:
      'Petr '}
2
3 # Vytvorime si kopii
4 for id, jmeno in studenti.items():
5     print(id, jmeno)
```

Příklad s Else, Break, Continue Statementy

- `else` vykoná se po skončení cyklu (není vykonán v případě volání `break`)
- `break` ukončí nejbližší cyklus `for` nebo `while`
- `continue` přejde na další interakci v cyklu

```
1  # Priklad s else statementem
2  for i in range(5):
3      print(i)
4  else:
5      print("Cyklus uspesne dokoncen")
6
7  # Priklad s break a continue statementy
8  for j in range(10):
9      if j == 3:
10         continue
11     if j == 7:
12         break
13  print(j)
14
```

Cyklus while

Python While cyklus

Příkaz `while` se používá pro opakované provádění, dokud je výraz pravdivý:

```
while podminka:
    telo
```

Příklady použití:

```
# Provadej vykonavani dokud pocet je mensi nez 10
pocet = 0
while pocet < 10:
    print(pocet)
    pocet += 2

# Pouziti break ve while loopu
pocet = 0
while pocet < 10:
    if pocet == 8:
        break
    print(pocet)
    pocet += 1
```

Python While cyklus

Příklady použití:

```
# Pouziti continue ve while loopu
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue
    print(count)
```

Výpočet odmocniny

Heronova metoda, také známá jako Babylónská metoda, je algoritmus pro výpočet druhé odmocniny čísla. Je založena na iterativní aproximaci a je definována následovně:Pro dané číslo S, pro které chceme najít druhou odmocninu, a počáteční odhad x_0, je iterativní vzorec pro Heronovu metodu:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

Metoda pokračuje v iteracích s tímto vzorcem, dokud je rozdíl mezi x_n^2 a S menší než určená tolerance ϵ. Konečná hodnota x je potom považována za aproximaci druhé odmocniny z S.

Heronova metoda pro Výpočet Druhé Odmocniny

Algorithm 1 Heronova Metoda

1: **procedure** odmocnina(S)
2: $x \leftarrow S$
3: **while** $|x^2 - S| > \epsilon$ **do**
4: $x \leftarrow \frac{1}{2}\left(x + \frac{S}{x}\right)$
5: **end while**
6: **return** x
7: **end procedure**

Heronova metoda pro Výpočet Druhé Odmocniny

```
1 import math
2
3 cislo = 121
4 x = cislo
5
6 while abs(x*x - cislo) > 1e-6:
7     x = 0.5*(x + cislo/x)
8
9 print(x)
```

Výpočet funkce `sin`

Taylorův rozvoj pro funkci sinus je dán vztahem:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Například, pro aproximaci $\sin(1)$ pomocí Taylorova rozvoje kolem $x = 0$ (Maclaurinovy řady):

$$\sin(1) \approx 1 - \frac{1^3}{3!} + \frac{1^5}{5!} - \frac{1^7}{7!} + \cdots$$

Výpočet funkce sin

```
1  import math
2
3  x = 0.7
4
5  suma = x
6  cislo = 1
7  prirustek = x
8  while abs(prirustek) > 1e-8:
9          prirustek *= -1*x*x / ((cislo + 1)*(cislo + 2))
10         cislo += 2
11         suma += prirustek
12
13 print(suma)
14 print(math.sin(0.7))
15 print("Difference",(suma - math.sin(0.7)))
```

Příkaz `match`

- Příkaz `match` je podobný konstrukci `case` a `switch` v jazyčích C, C++, C++či Java.
- Příkaz `match` byl představen v Pythonu 3.10 k zjednodušení podmíněné logiky.
- Příkaz `match` umožňuje porovnávat hodnotu s různými vzory, pouze první porovnání které vyhoví bude "vykonáno"

```
1          match hodnota:
2              case vzor1:
3                  # kodovy blok
4              case vzor2:
5                  # kodovy blok
6
```

- Zlepšuje čitelnost a snižuje zanořené `if` podmínky.

Příkaz match

- Při použití symbolu _ je možno zachytit všechny hodnoty.
- S pomocí symbolu | je možné porovnat hodnoty s několika možnostmi.

```
ovoce = "cokoli"  # jablko, banan, hruska, svestka
                  # tresen, merunka, jahoda,
                  # ostruzina, malina

match ovoce:
    case "jablko" | "hruska":
        print("jadroviny")
    case "svestka" | "tresen" | "merunka":
        print("peckoviny")
    case _:
        print("Vsechno ostatni")
```

Přednáška 5

Funkce

Funkce v Pythonu

- Funkce je uspořádané pořadí příkazů, provádějících požadovanou operaci pro zadaný výčet parametrů. Příkazy jsou deklarovány v těle funkce, parametry jsou uvedeny v záhlaví funkce.
- Funkce v Pythonu se definuje pomocí klíčového slova `def`, následovaného názvem funkce a parametry v závorkách, funkce nemusí mít žádné parametry `()`.
- Tělo funkce následuje na dalším řádku a musí být odsazeno.
- Syntaxe pro definování funkce vypadá následovně:

```
1 def nazev_funkce(parametry_funkce):    # jmeno funkce a
    parametry funkce
2     """
3     docstring - dokumentacni retezec
4     """
5     prikazy                            # prikazy tela
    funkce
```

Funkce v Pythonu — Fibonacciho čísla

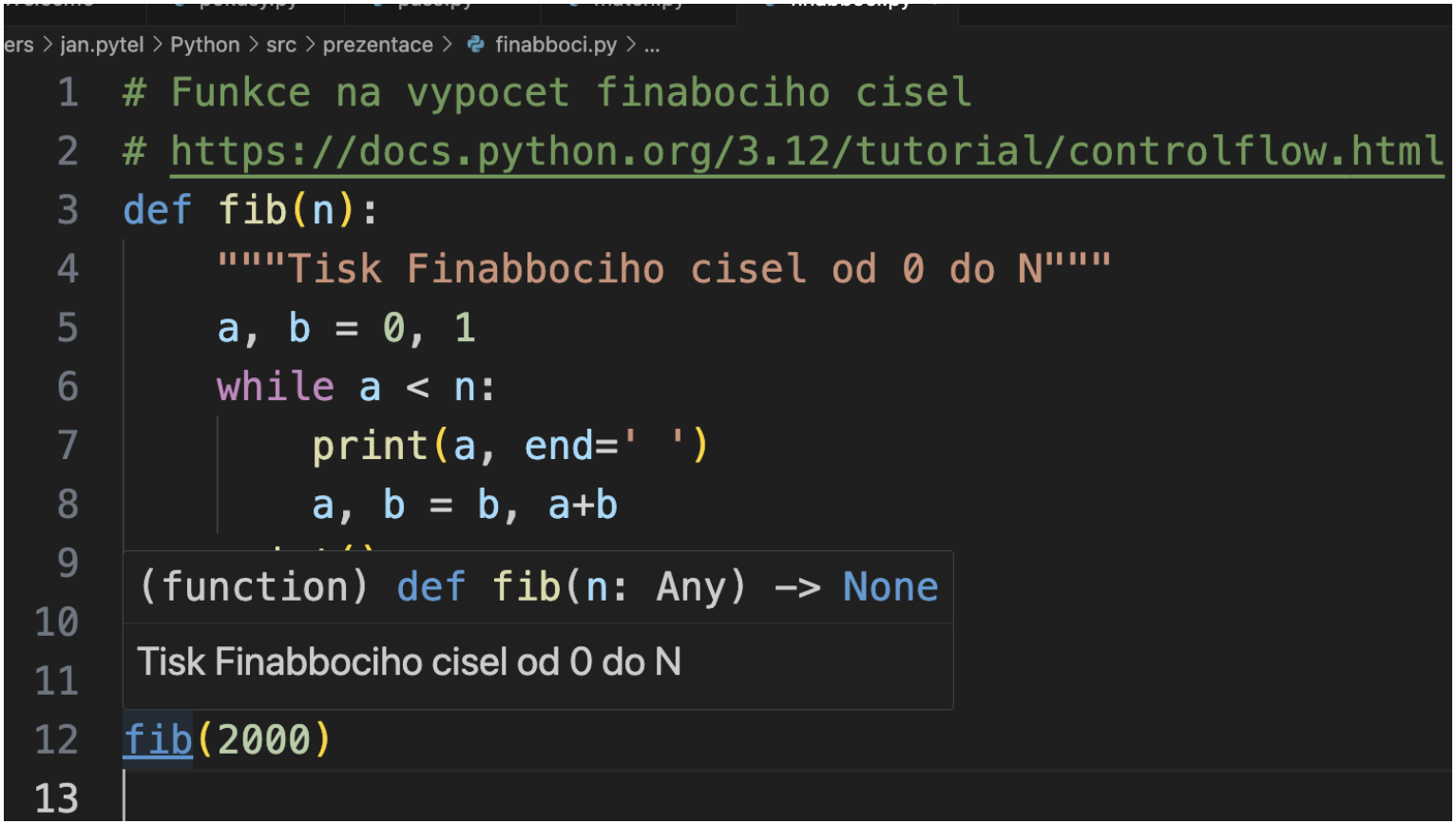
- Fibonacciho posloupnost je série čísel, kde každé číslo je součtem dvou předcházejících čísel, obvykle začínající na 0 a 1. Fibonacciho posloupnost lze definovat rekurentním vztahem:

$$F(n) = F(n-1) + F(n-2), \text{ pro } n > 1, \quad F(0) = 0, F(1) = 1$$

```
1  # Funkce na vypocet finabociho cisel
2  # https://docs.python.org/3.12/tutorial/controlflow.html
3  def fib(n):
4      """Tisk Fibonacciho cisel od 0 do N"""
5      a, b = 0, 1
6      while a < n:
7          print(a, end=' ')
8          a, b = b, a+b
9      print()
10
11 # Zavolani funkce s parametrem 2000
12 fib(2000)
13 print(fib)
```

Funkce v Pythonu

- Prvním příkazem těla funkce může být volitelně řetězcový literál; tento řetězcový literál je dokumentační řetězec funkce neboli docstring.



Funkce v Pythonu

- Spuštění funkce zavádí novou tabulku symbolů, která se používá pro lokální proměnné funkce.
- Funkce může mít parametry, které jí slouží k přijímání hodnot pro zpracování.
- Parametry volání funkce jsou zavedeny do lokální tabulky symbolů volané funkce při jejím volání; argumenty se tedy předávají pomocí volání podle hodnoty (kde hodnota je vždy odkaz na objekt, nikoli hodnota objektu).

```
1 def soucetTriCisel(cislo1, cislo2, cislo3):
2     """Toto je trivialni funkce scitajici tri cisla"""
3     soucet = cislo1 + cislo2 + cislo3
4     return soucet
5
6 vysledek = soucetTriCisel(10,20,30)
7 print(vysledek)
```

Funkce v Pythonu

- Definice funkce spojuje jméno funkce s objektem funkce v aktuální tabulce symbolů. Interpret rozpozná objekt, na který toto jméno ukazuje, jako uživatelsky definovanou funkci.

```
>>>
fib
<function fib at 0x1024c4e00>
f = fib
f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Návratové Hodnoty Funkcí v jazyku Python

- Funkce v Pythonu mohou vracet hodnoty pomocí příkazu `return`, slouží k ukončení funkce a vrácení hodnoty.
- Příkaz `return` je následován hodnotou nebo výrazem, který se má vrátit.

```
1 # Funkce s parametrem
2 def secti(a, b):
3     return a + b
4
5 print(secti(3, 5))
```

- Funkce v Pythonu mohou vracet více hodnot jako n-tici.

```
1 def obvod_a_obsah_kruhu(polomer):
2     obvod = 2 * 3.14 * polomer
3     obsah = 3.14 * (polomer ** 2)
4     return obvod, obsah
5
6 output = obvod_a_obsah_kruhu(5)
7 print(f"Obvod: {output[0]}, obsah: {output[1]}")
```

Funkce v jazyku Python — Použití Vracených Hodnot

- Jestliže není použit žádný příkaz `return`, funkce vrací v Pythonu implicitně `None`.

```
1 def pozdrav(slovo):
2     """Funkce ktera tiskne <<<<< slovo >>>>>"""
3     print("<<<<< ",slovo, ">>>>>")
4
5 pozdrav("Ahoj")
6 print(pozdrav("Ahoj"))
```

```
1 <<<<<  Ahoj >>>>>
2 <<<<<  Ahoj >>>>>
3 None
```

- Vracená hodnota může být přiřazena do proměnné nebo použita přímo.

```
1         vysledek = ziskej_hodnoty()
2         print(vysledek)
```

Fibonacciho cisla v Pythonu vracející seznam čísel

- Přepsání předchozí funkce `fib` tak, aby vracela list hodnot Fibonacciho posloupnosti:

```
1  # Funkce na vypocet finabociho cisel
2  # https://docs.python.org/3.12/tutorial/controlflow.html
3  def fib(n):
4      """Vraceni Fibonacciho cisel od 0 do N v seznamu"""
5      seznam = []
6      a, b = 0, 1
7      while a < n:
8          seznam.append(a)
9          a, b = b, a+b
10     return seznam
11
12 # Zavolani funkce s parametrem 2000
13 print(fib(2000))
```

Funkce v jazyku Python — defaultní parametry ve funkcích

- Defaultní parametry funkcí umožňují přiřadit výchozí hodnotu parametru, pokud není hodnota argumentu předána během volání funkce.
- Při volání funkce pokud není přítomen defaultní parametr, funkce je spuštěna bez problému.
- Po parametru/parametrech s defaultní hodnotou nemůže následovat parametr bez defaultní hodnoty

```
1 def pozdrav(prijmeni, jmeno=""):
2     print("Dobry den, jsi: ", prijmeni + " " + jmeno)
3
4
5 pozdrav("Capek", "Karel")  # Dobry den, jsi: Capek Karel
6 pozdrav("Novak")           # Dobry den, jsi: Novak
```

- Výchozí hodnota se vyhodnotí pouze jednou. To je rozdíl, pokud je výchozí hodnotou proměnlivý objekt, jako je seznam, slovník nebo instance většiny tříd.

Funkce v jazyku Python — vyhodnocení defaultní parametry ve funkcích

```
1  cislo = 3
2  def faktorial(n = cislo):
3      suma = 1
4      for i in range(1, n+1):
5          suma *= i
6      return suma
7
8  seznam = [5]
9  def faktorial_ze_seznamu(n = seznam):
10     suma = 1
11     cislo = seznam[0]
12     for i in range(1, cislo+1):
13         suma *= i
14     return suma
15
16 cislo = 10
17 seznam[0] = 25
18 print(faktorial())                  # 6
19 print(faktorial_ze_seznamu())  # 155112100433309859840000000
```

Funkce v jazyku Python — více o parametrech a argumentech

- Defaultní parametry funkcí umožňují přiřadit výchozí hodnotu parametru, pokud není hodnota argumentu předána během volání funkce.
- Při volání funkce pokud není přítomen defaultní parametr, funkce je spuštěna bez problému.
- Po parametru/parametrech s defaultní hodnotou nemůže následovat parametr bez defaultní hodnoty

```
1 def pozdrav(prijmeni, jmeno=""):
2     print("Dobry den, jsi: ", prijmeni + " " + jmeno)
3
4
5 pozdrav("Capek", "Karel")  # Dobry den, jsi: Capek Karel
6 pozdrav("Novak")           # Dobry den, jsi: Novak
```

- Výchozí hodnota se vyhodnotí pouze jednou. To je rozdíl, pokud je výchozí hodnotou proměnlivý objekt, jako je seznam, slovník nebo instance většiny tříd.

Funkce v jazyku Python — více o parametrech a argumentech

- Parametry a argumenty je možno kategorizovat:
  - **poziční parametry** — záleží pouze na pozici v které daný parametr je napsán
  - **keyword arguments** — vztah mezi hodnotou a parametrem
  - **defaultní parametry** — viz předchozí text
- Pomocí znaků * a / je možno vynucovat které typy argumentů funkce akceptuje

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------    ----------     ----------
        |              |                  |
        |        Positional or keyword    |
        |                                 - Keyword only
         -- Positional only
```

Funkce v jazyku Python — více o parametrech a argumentech

```
1  def fce(cislo1, cislo2, cislo3):
2      return cislo1 + cislo2*10 + cislo3*100
3
4  # print(fce(10, 20)) #fce() missing 1 required positional
       argument: 'cislo3'
5  print(fce(10, 20, 30)) # Vysledek 3210
6  print(fce(cislo1 = 10, cislo2 = 20, cislo3 = 30)) #  3210
7  print(fce(cislo2 = 20, cislo3 = 30, cislo1 = 10)) #  3210
8  # print(fce(cislo1 = 10, 20, 30)) # Positional argument
       follows keyword argument
9
10 def fce1(cislo1, /, cislo2, cislo3):
11     return cislo1 + cislo2*10 + cislo3*100
12
13 print(fce1(10, 20, 30)) # fce1() takes 0 positional
       arguments but 3 were given
14 print(fce1(10, cislo2 = 20, cislo3 = 30)) # 3210
```

Funkce v jazyku Python — více o parametrech a argumentech

```
1 # print(fce1(cislo2 = 20, cislo3 = 30, cislo1 = 10)) #fce1()
      got some positional-only arguments passed as keyword
    arguments: 'cislo1'
2
3 def fce2(*, cislo1, cislo2, cislo3):
4     return cislo1 + cislo2*10 + cislo3*100
5
6 # print(fce2(10, 20, 30)) # Chyba - fce2() takes 0
    positional arguments but 3 were given
7 print(fce2(cislo1 = 10, cislo2 = 20, cislo3 = 30)) # 3210
8 print(fce2(cislo2 = 20, cislo3 = 30, cislo1 = 10)) # 3210
```

Funkce v jazyku Python — parametry a argumenty *argv

- Arbitrary Argument Lists v Pythonu umožňují funkcím přijímat libovolný počet argumentů.
- Unpacking v Pythonu je proces, kdy se sekvence (např. seznam nebo tuple) rozbalí na jednotlivé hodnoty.

```
1  def suma(*cisla):
2      """Soucet vsech cisel na vstupu"""
3      total = 0
4      for i in cisla:
5          total += i
6
7      return i
8
9  print(suma(1,2,3,4,5))
10 print(suma(*[10, 20, 30, 40, 50]))
11 print(suma(*(5, 6, 7, 8)))
```

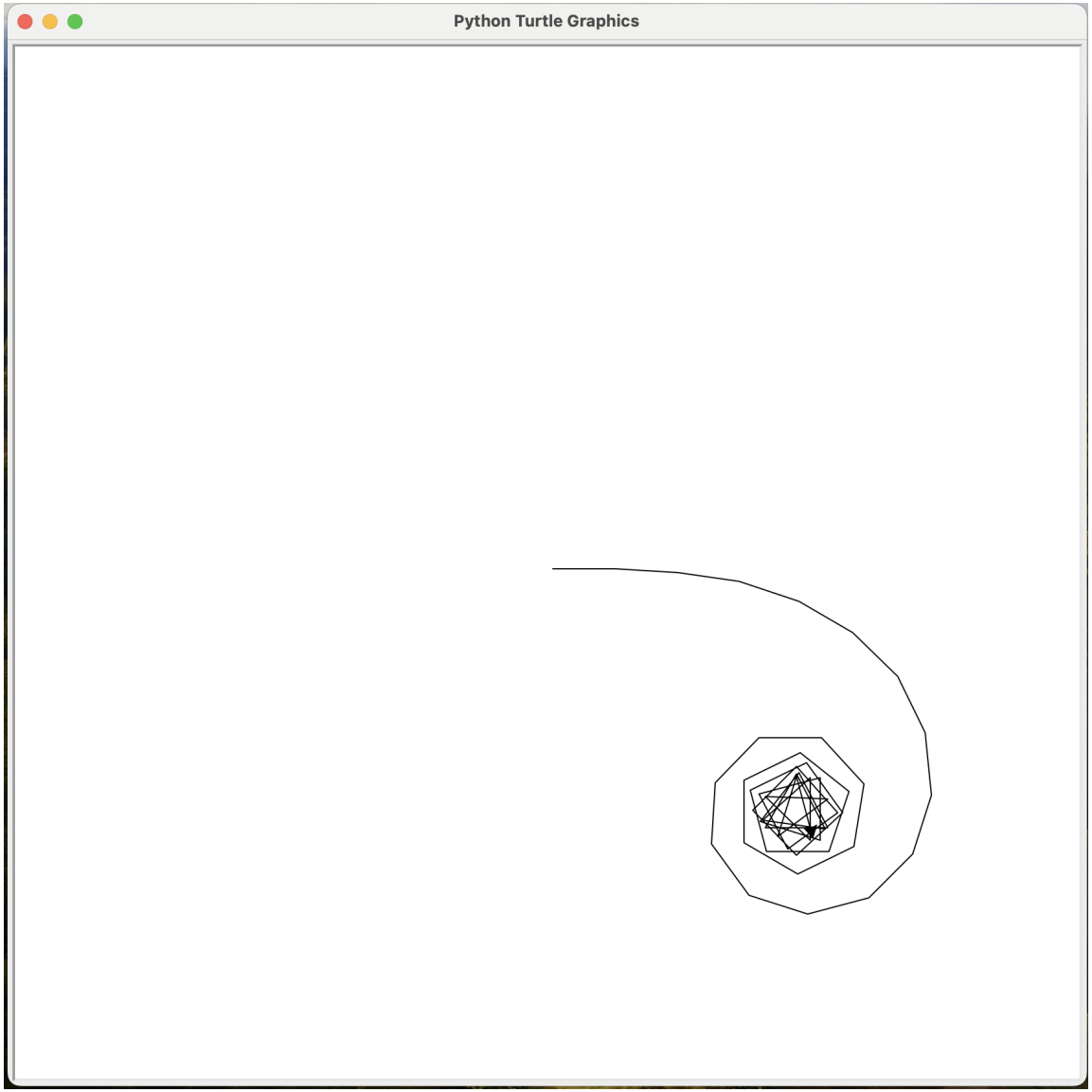
Základní kreslení obrázků s Turtle

Python Turtle: Úvod

- Modul Turtle v Pythonu umožňuje kreslení pomocí želvy známé z Logo.
- Základní princip: vytvoření želvy, která se pohybuje a kreslí na plátně pomocí jednoduchých příkazů.
- Více informací na `https://docs.python.org/3/library/turtle.html`

```
1 import turtle # import zelvy
2
3 zelva = turtle.Turtle() # Vytvoreni zelvy
4 for i in range(50):
5     zelva.right(i*3)
6     zelva.forward(50)
7
8 zelva.screen.mainloop()
```

Grafický výsledek



Základní Příkazy s Python Turtle

Příkaz	Popis
`forward(vzdálenost)`	Posune želvu vpřed o zadanou vzdálenost.
`backward(vzdálenost)`	Posune želvu vzad o zadanou vzdálenost.
`left(úhel)`	Otočí želvu doleva o zadaný úhel.
`right(úhel)`	Otočí želvu doprava o zadaný úhel.
`setpos(x, y)`	Nastaví pozici želvy na zadané souřadnice x a y.
`goto(x, y)`	Přesune želvu na nové místo s kreslením.
`teleport(x, y)`	Přesune želvu na nové místo bez kreslení.
`setx(x)`	Nastaví pozici želvy na danou souřadnici x.
`sety(y)`	Nastaví pozici želvy na danou souřadnici y.
`home()`	Posune želvu na počáteční pozici a směr.

Pokročilé Příkazy s Python Turtle

Příkaz	Popis
`circle(poloměr)`	Nakreslí kruh s daným poloměrem.
`dot(velikost, barva)`	Vykreslí tečku s danou velikostí a barvou.
`speed(rychlost)`	Nastaví rychlost pohybu želvy.
`fillcolor(barva)`	Nastaví barvu výplně tvarů.
`pencolor(barva)`	Nastaví barvu pera.
`bgcolor(barva)`	Nastaví barvu pozadí.
`pendown()`	Sestoupí pero a začne kreslit.
`penup()`	Zvedne pero a přesune želvu bez kreslení.
`clear()`	Smaže všechny stopy želvy na obrazovce.
`fill(True/False)`	Zapne nebo vypne výplň objektů.
`begin_fill()`	Začne kreslit tvar pro vyplnění.
`end_fill()`	Ukončí kreslení tvaru pro vyplnění.
`isdown()`	Vrátí True, pokud je pero dolů, jinak False.

Pokročilé Příkazy s Python Turtle

Příkaz	Popis
`isvisible()`	Vrátí True, pokud je želva viditelná, jinak False.
`showturtle()`	Zobrazí želvu na obrazovce.
`hideturtle()`	Skryje želvu na obrazovce.
`setheading(úhel)`	Nastaví směr, do kterého bude želva směřovat.
`turtle.distance(x, y)`	Vrátí vzdálenost želvy od bodu s danými souřadnicemi.
`turtle.heading()`	Vrátí aktuální úhel želvy na obrazovce.
`turtle.towards(x, y)`	Vrátí úhel mezi aktuální pozicí želvy a bodem s danými souřadnicemi.
`turtle.reset()`	Vrátí všechny nastavení želvy na výchozí hodnoty.

Obrazovka Python Turtle

- Obrazovka Python Turtle slouží jako plátno, na kterém želva kreslí.
- Poskytuje vizuální reprezentaci pohybů želvy a výkresů.
- Můžeme ovládat velikost, barvu pozadí a další vlastnosti obrazovky Turtle.

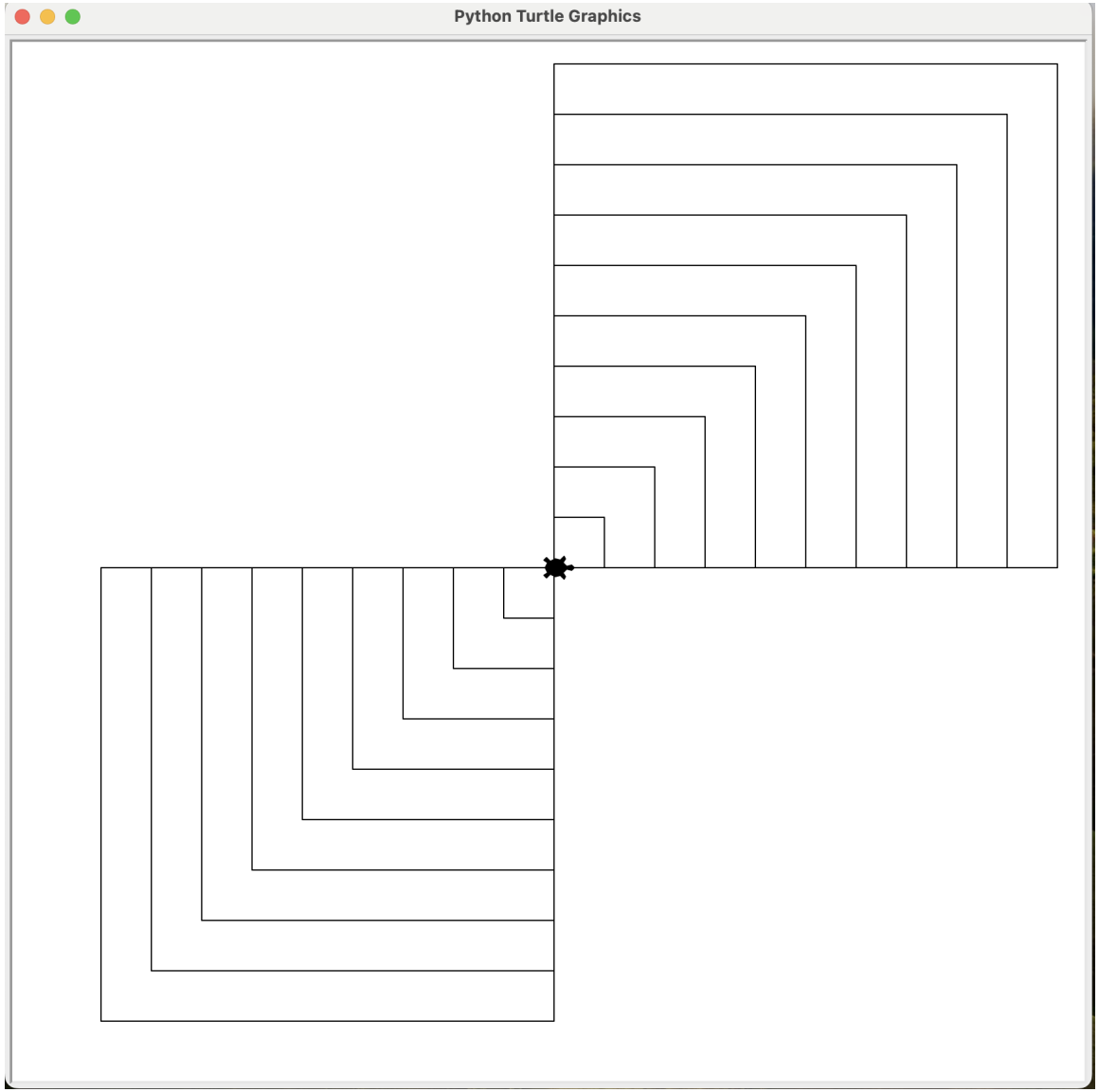
Funkce Obrazovky Turtle:

`setup(šířka, výška)`	Nastavení velikosti obrazovky
`bgcolor(barva)`	Nastavení barvy pozadí
`hideturtle()`	Skrytí želvy na obrazovce
`showturtle()`	Zobrazení želvy na obrazovce
`mainloop()`	Funkce pro udržení programu otevřeného až do uzavření okna.

Python Turtle - Příklad 1

```
1  import turtle # import zelvy
2
3  zelva = turtle.Turtle() # Vytvoreni zelvy
4  zelva.shape('turtle')
5  posun = 400
6  for _ in range(20):
7      for _ in range(4):
8          zelva.forward(posun)
9          zelva.left(90)
10     posun -= 40
11
12 zelva.screen.mainloop()
```

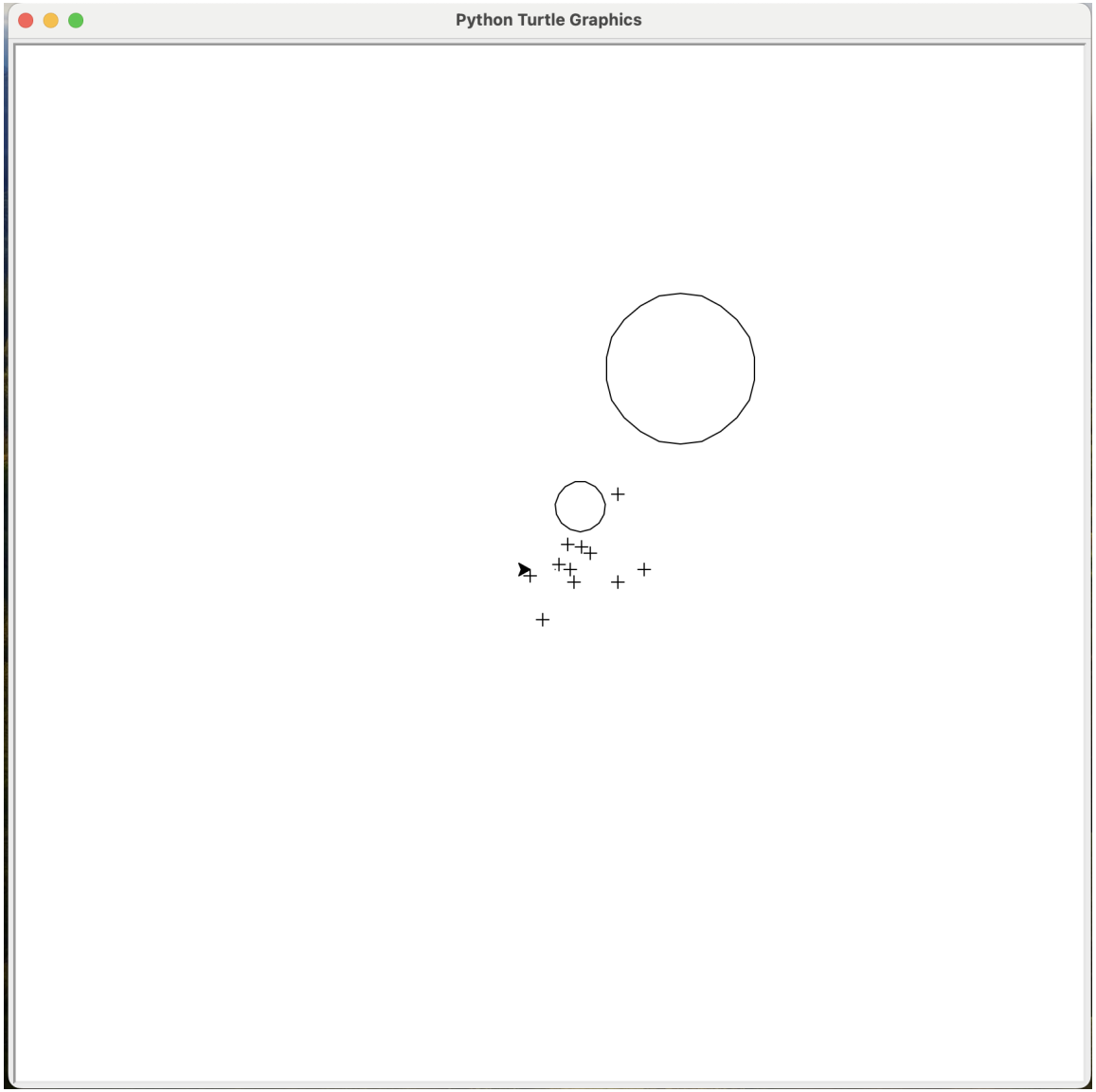
Python Turtle - Příklad 1 výsledek



Python Turtle - Příklad 1

```
1  import turtle # import zelvy
2
3  zelva = turtle.Turtle() # Vytvoreni zelvy
4  zelva.shape('turtle')
5  posun = 400
6  for _ in range(20):
7      for _ in range(4):
8          zelva.forward(posun)
9          zelva.left(90)
10     posun -= 40
11
12 zelva.screen.mainloop()
```

Python Turtle - Grafické znázornění úlohy 2.3



Rekurze

Rekurze v Python funkcích

- Rekurze je programovací technika, kdy funkce volá sám sebe k řešení menších instancí stejného problému.
- Rekurze spočívá v řešení problémů rozdělením na menší podobné podproblémy.
- Zásadní je zabránit nekonečné rekursi.
- Rekurze může být elegantním řešením pro určité problémy, ale nemusí být vždy nejefektivnější.
- Hloubka rekurze — počet rekurzivních volání funkce.

```
1 def rekurze(parametry):
2     # blok kodu
3     if podminka:
4         # neco co ukonci rekurzi
5     else:
6         rekurze(parametry)
7     # blok kodu
```

Přímá a nepřímá rekurse, výhody a nevýhody rekurze

- **Existují dva typy rekurze**:
  - **Přímá rekurze** nastává, když funkce volá sama sebe přímo.
  - **Nepřímá rekurze** je situace, kdy funkce volá jinou funkci, která zase může volat původní funkci.
- **Výhody Rekurze:**
  - Jednoduché a čisté řešení problémů.
  - Zlepšuje čitelnost kódu pro určité problémy.
  - Často odpovídá přirozené rekurzivní povaze problémů.
  - Může být krátký a elegantní zápis řešení.
- **Nevýhody Rekurze:**
  - Může využívat více paměti než iterativní řešení.
  - Riziko přetečení zásobníku při hluboké rekurzi.
  - Může být méně efektivní než iterativní řešení pro některé problémy.
  - Obtížnější k pochopení pro některé vývojáře.

Iterace

- Opakování činnosti s cyklem nebo smyčkou
- Často používané pro zpracování seznamů a lineárních operací
- Může být efektivnější pro některé problémy s lineárním průchodem
- Je zde velmi úzká návaznost s rekurzí
- Rekurzi je možno přepsat iterací a naopak
- **Výhody Iterace:**
  - Efektivní pro lineární zpracování dat.
  - Méně paměťově náročné než rekurze.
  - Jednodušší a přehlednější pro některé problémy.
- **Nevýhody Iterace:**
  - Pro některé problémy může být složitější než rekurze.
  - Může vyžadovat více kódu a být méně elegantní.
  - Méně intuitivní pro některé matematické problémy.

Rekurze v Python funkcích — Výpočet faktoriálu rekurzí

- Faktoriál čísla n má definici

$$n! = \begin{cases} 1 & , \text{když } n = 0 \\ n \times (n-1)! & , \text{když } n > 0 \end{cases}$$

- Tento kód demonstruje přímou rekurzi při výpočtu faktoriálu čísla.

```
1 def faktorial(n):
2     if n <= 1:
3         return 1
4     else:
5         return n * faktorial(n-1)
```

Rekurze v Python funkcích — Fibonacciho posloupnosti

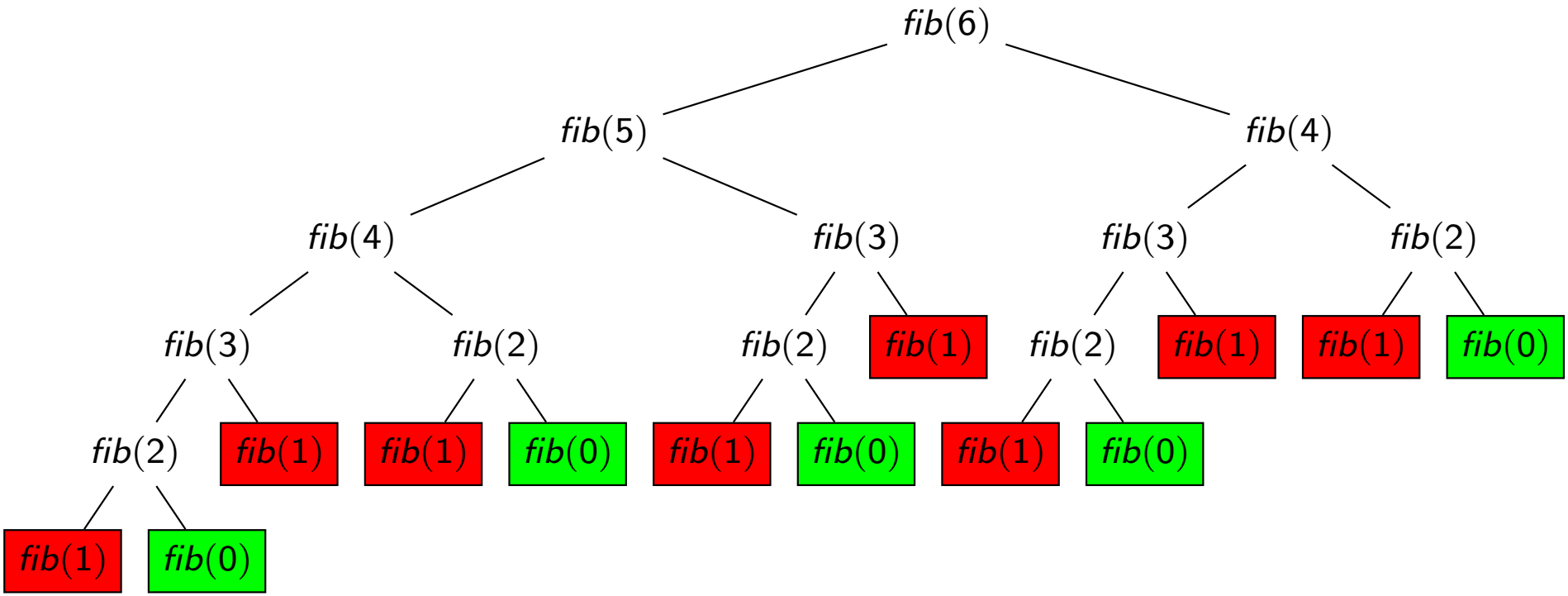
- Fibonacciho posloupnost má následující definici

$$F(n) = \begin{cases} 0 & ,\text{když } n = 0 \\ 1 & ,\text{když } n = 1 \\ F(n-1) + F(n-2) & ,\text{když } n > 1 \end{cases}$$

- Tento kód demonstruje přímou rekurzi při výpočtu Fibonacciho posloupnosti (výsledný kód je značně neefektivní)

```
1 def fibonacci(n):
2   if n <= 1:
3       return n
4   else:
5       return(fibonacci(n-1) + fibonacci(n-2))
```

Fibonacci posloupnost - strom volání rekurzivní funkce fib(6)



Rekurze v Python funkcích — Fibonacciho posloupnosti

```
1  import time
2
3  pocet_volani  = 0
4
5  def fibonacci(n):
6    global pocet_volani
7    pocet_volani += 1
8    if n <= 1:
9        return n
10   else:
11       return(fibonacci(n-1) + fibonacci(n-2))
12
13 for i in range(1,50):
14     pocet_volani = 0
15     # Zjisti aktualni cas v sekundach
16     curr = time.time()
17     fib = fibonacci(i)
18     diff = time.time() - curr
19     print(f"{i}: {fib} volani = {pocet_volani} "
20           f"cas[sec] = {diff:0.2f}")
```

Rekurze v Python funkcích — Fibonacciho posloupnosti

n	výsledek	počet volání	čas[sec]
1	1	1	0.00
2	1	3	0.00
3	2	5	0.00
4	3	9	0.00
5	5	15	0.00
6	8	25	0.00
7	13	41	0.00
8	21	67	0.00
9	34	109	0.00
10	55	177	0.00
11	89	287	0.00
12	144	465	0.00
13	233	753	0.00
14	377	1219	0.00

Rekurze v Python funkcích — Fibonacciho posloupnosti

n	výsledek	počet volání	čas[sec]
15	610	1973	0.00
16	987	3193	0.00
17	1597	5167	0.00
18	2584	8361	0.00
19	4181	13529	0.00
20	6765	21891	0.00
21	10946	35421	0.00
22	17711	57313	0.00
23	28657	92735	0.01
24	46368	150049	0.01
25	75025	242785	0.01
26	121393	392835	0.02
27	196418	635621	0.04
28	317811	1028457	0.06

Rekurze v Python funkcích — Fibonacciho posloupnosti

n	výsledek	počet volání	čas[sec]
29	514229	1664079	0.09
30	832040	2692537	0.15
31	1346269	4356617	0.24
32	2178309	7049155	0.38
33	3524578	11405773	0.62
34	5702887	18454929	1.01
35	9227465	29860703	1.61
36	14930352	48315633	2.62
37	24157817	78176337	4.26
38	39088169	126491971	6.96
39	63245986	204668309	11.32
40	102334155	331160281	18.33
41	165580141	535828591	29.64
42	267914296	866988873	47.80

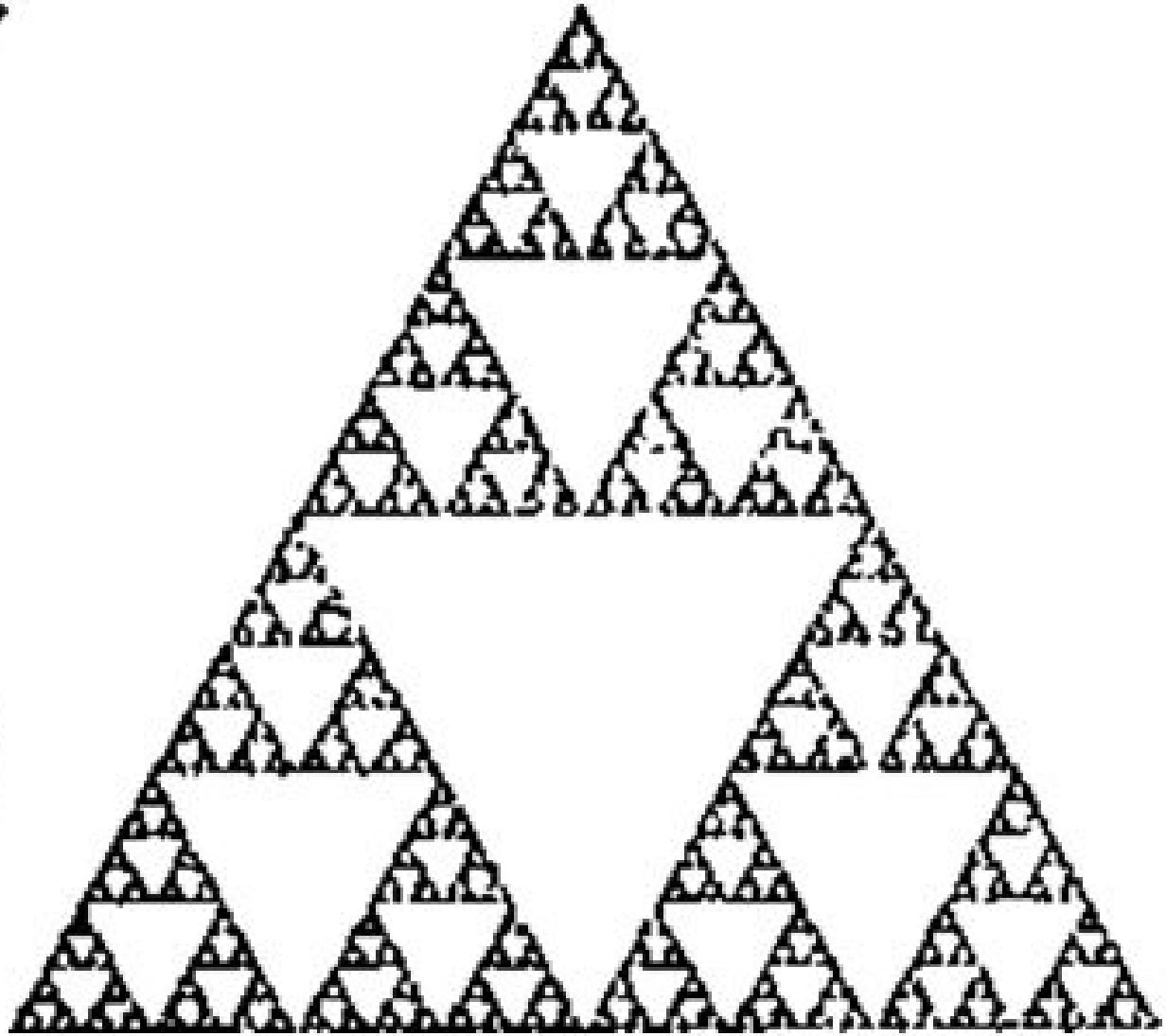
Rekurze v Python funkcích — Fibonacciho posloupnosti

n	výsledek	počet volání	čas[sec]
43	433494437	1402817465	77.39
44	701408733	2269806339	122.43
45	1134903170	3672623805	198.14
46	1836311903	5942430145	321.02
47	2971215073	9615053951	725.29
48	4807526976	15557484097	869.81
49	7778742049	25172538049	2463.32

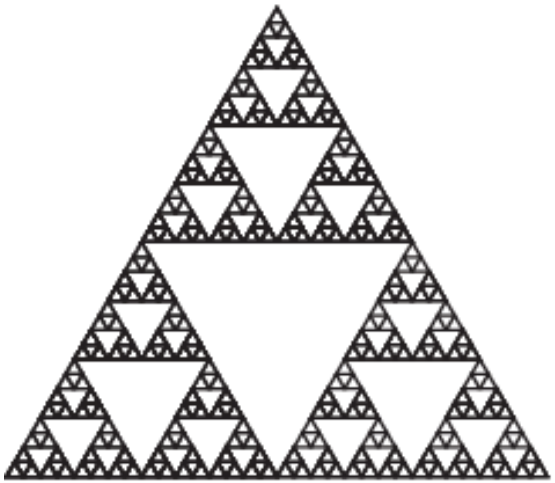
Čas nutný pro výpočet Fibonacciho posloupnosti pomocí interakce pro prvních 49 prvků je 37µs

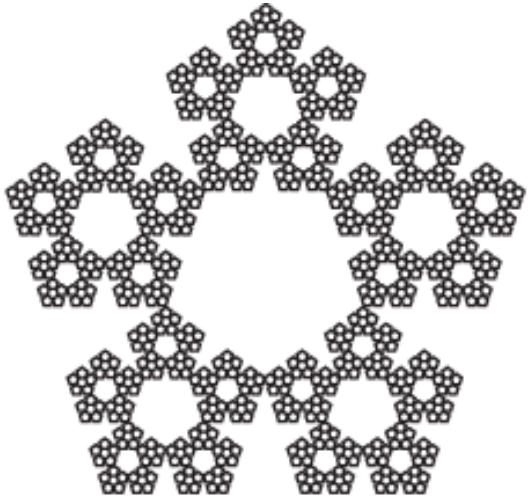
Ukázka fraktálů - nekonečně členité útvary

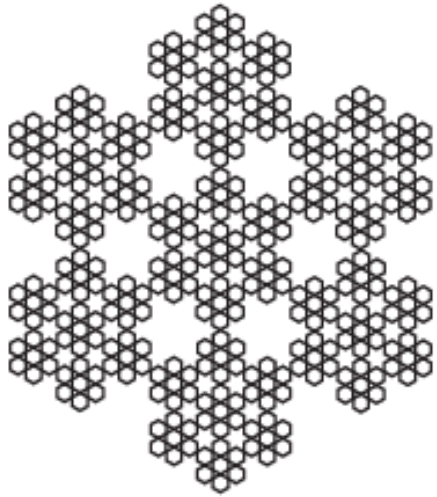
Ukázka fraktálů - 1

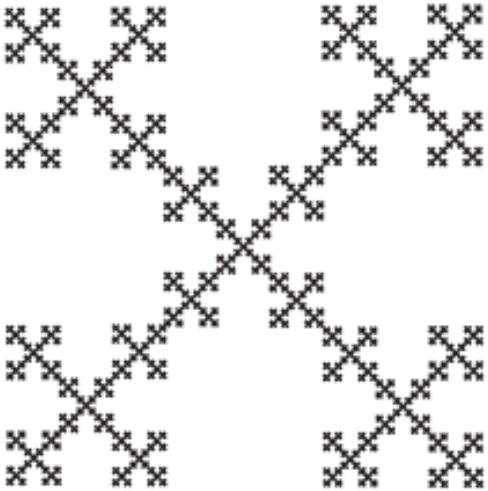


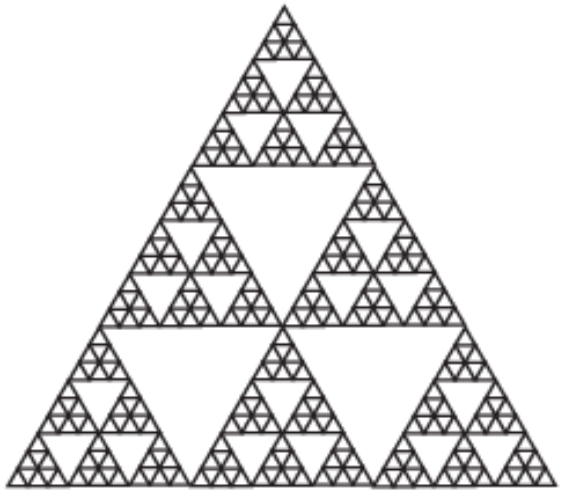
Ukázka fraktálů - 2

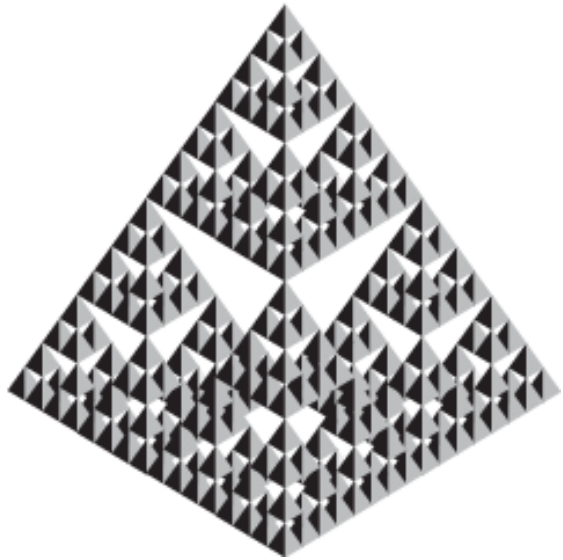




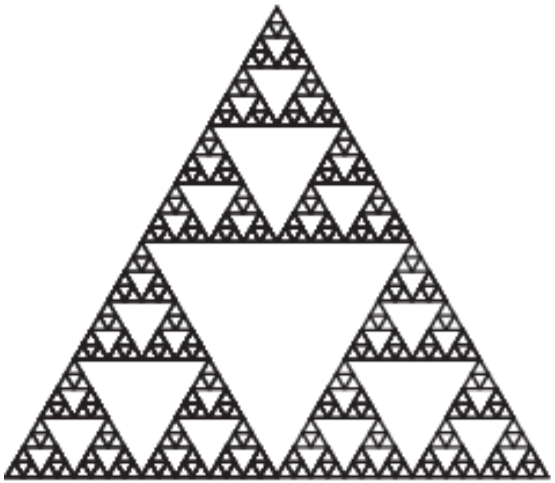
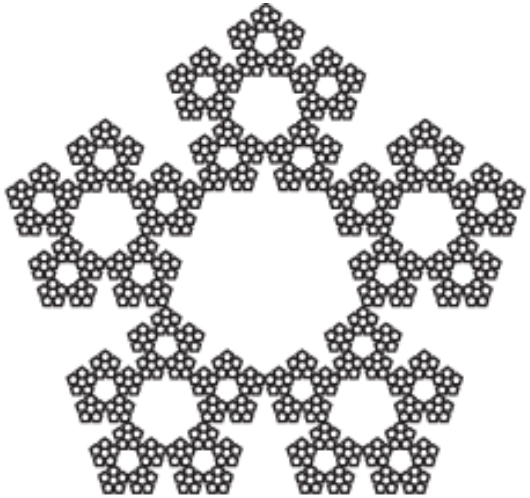
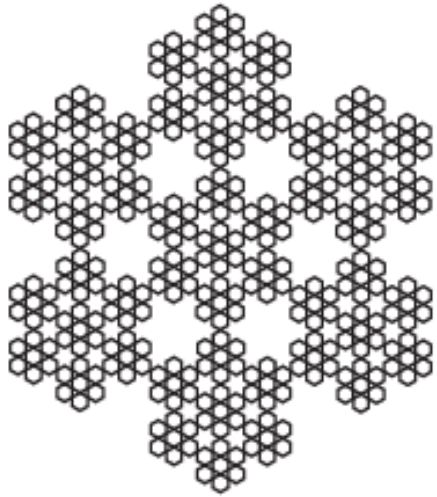
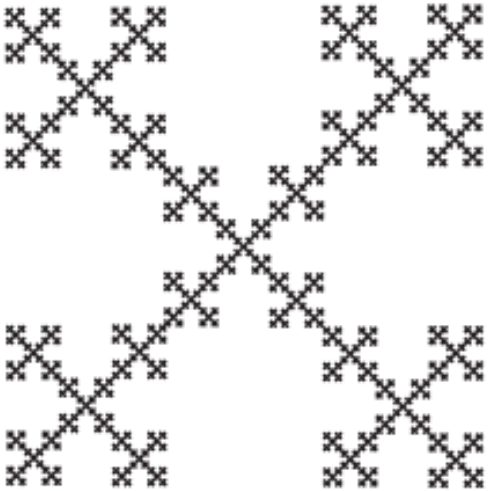
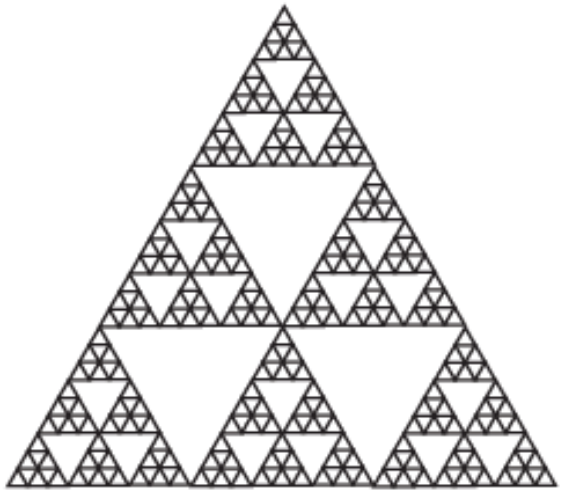
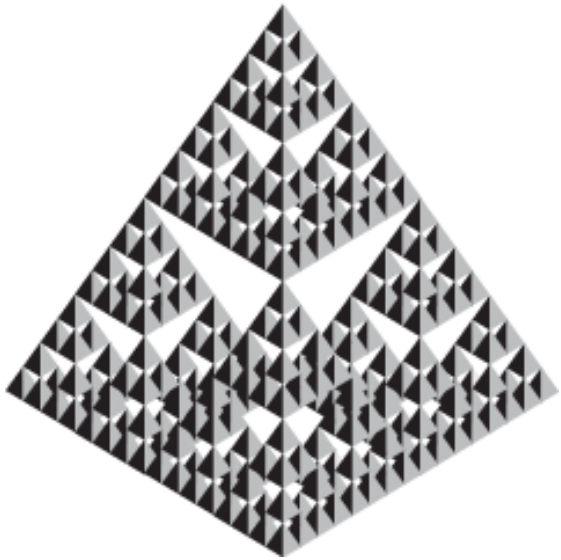








Ukázka fraktálů - 3 - Kochova vločka

List comprehenions

Co je List Comprehension?

- List Comprehension je elegantní způsob tvorby seznamů v Pythonu.
- Umožňuje kompaktní a čistý zápis operací na seznamech.
- Zlepšuje efektivitu a čitelnost kódu.
- Základní syntaxe List Comprehension:

```
[expression for item in list if condition]
[x**2 for x in range (1 ,6) ]
```

- Podporuje vytváření nových seznamů, slovníků nebo množin.
- Kompaktní formát umožňuje provádět operace na seznamech efektivně.

Příklady List Comprehension

- Seznam druhých mocnin čísel od 1 do 5:

```
1 [x**2 for x in range(1,6)]
```

- Seznam lichých čísel od 1 do 10:

```
1 [x for x in range(1,11) if x % 2 != 0]
```

- Seznam písmen v řetězci `Dobry den`:

```
1 [letter for letter in 'Dobry den']
```

- Seznam součtů prvků z dvou seznamů:

```
1 [x + y for x in [1, 2, 3] for y in [4, 5, 6]]
```

Další Příklady List Comprehension

- Výběr x z listu (x,y)

```
x = [x for x, y in body]
y = [y for x, y in body]
```

- Vrať (x,y) pro body (x,y) které jsou do určité vzdálenosti od těžiště:

```
[(x, y) for x,y in body if vzdal((x, y), teziste) < 20]
```

- Seznam platných emailových adres z daného seznamu:

```
[email for email in emails if is_valid(email)]
```

- Seznam slov s délkou větší než 6 z daného seznamu slov:

```
[sport for sport in ['biatlon', 'lyzovani', 'cyklistika'
    , 'beh'] if len(sport) > 5]
```

List Comprehension

```
>>> print([x**2 for x in range(1,6)])
[1, 4, 9, 16, 25]
>>> print([x for x in range(1,11) if x % 2 != 0])
[1, 3, 5, 7, 9]
>>> print([letter for letter in 'Dobry den'])
['D', 'o', 'b', 'r', 'y', ' ', 'd', 'e', 'n']
>>> print([x + y for x in [1, 2, 3] for y in [4, 5, 6]])
[5, 6, 7, 6, 7, 8, 7, 8, 9]
>>> body = [(10,20), (23,65), (-3,-20)]
>>> print([x for x, y in body])
[10, 23, -3]
>>> print([sport for sport in ['biatlon', 'lyzovani', '
    cyklistika', 'beh']
...         if len(sport) > 5])
['biatlon', 'lyzovani', 'cyklistika']
>>> print([(x, y) for x, y in body if x + y  < 20])
[(-3, -20)]
```

Simulace

Co je Monte Carlo Simulace?

- Monte Carlo simulace je výpočetní technika, která využívá náhodný výběr k řešení matematických problémů.
- Poskytuje přibližné řešení simuluje velké množství náhodných vzorků nebo scénářů.
- Metody Monte Carlo jsou široce využívány v různých oblastech jako je finance, inženýrství a fyzika pro analýzu rizika, optimalizaci a analýzu nejistoty.
- Monte Carlo simulace se skládá z následujících kroků:
  1. Definování problému a modelování systému.
  2. Generování náhodných vstupů na základě pravděpodobnostních rozdělení.
  3. Provedení simulací spuštěním modelu s náhodnými vstupy.
  4. Analýza výsledků a vyvození závěrů na základě statistické analýzy.

Monte Carlo Simulace v Pythonu — Úvod do modulu random

- Python poskytuje výkonné knihovny jako NumPy a pandas pro manipulaci s numerickými výpočty a analýzu dat.
- Modul ‘random‘ v Pythonu lze použít k generování náhodných čísel potřebných pro Monte Carlo simulace.
- Modul random v Pythonu poskytuje funkce pro generování náhodných čísel a sekvencí.
- Je to vestavěný modul, který je užitečný pro různé aplikace jako jsou simulace, hry a statistická analýza.
- Pro generování náhodného celého čísla v daném rozmezí pomocí `random.randint()`:

```
1 import random
2
3 nahodne_cislo = random.randint(1, 10)
4 print(nahodne_cislo)
```

Výběr Náhodného Prvku z Listu

- Pro výběr náhodného prvku z listu pomocí `random.choice()`:

```
1 import random
2
3 ovoce = ['jablko', 'banan', 'tresen']
4 nahodne_ovoce = random.choice(ovoce)
5 print(nahodne_ovoce)
```

- Pro náhodné zamíchání prvků v listu pomocí `random.shuffle()`:

```
1 import random
2
3 list = [1, 2, 3, 4, 5]
4 random.shuffle(list)
5 print(list)
```

Proč použít `random.seed`

- Funkce `random.seed` v Pythonu se používá k inicializaci generátoru náhodných čísel.
- Pomáhá stanovit výchozí bod pro generování náhodných čísel, což zajišťuje reprodukovatelnost výsledků.
- Využití `random.seed` se stejným základem zaručuje stejnou posloupnost náhodných čísel pokaždé.
- Pro nastavení specifické hodnoty základu: `random.seed(základ)`

```
1 import random
2
3 random.seed(42)
4 print(random.randint(1, 100))}
```

Úvod k Funkci random.normalvariate(strh, smero)

Gaussova funkce (normální distribuční funkce) je definována jako:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3}$$

kde:

- x je proměnná,
- μ je střední hodnota distribuce,
- σ je směrodatná odchylka distribuce,
- e je základ přirozeného logaritmu a
- π je matematická konstanta pi.
- Funkce `random.normalvariate(strh, smero)` v Pythonu se používá k generování náhodných čísel z normálního rozdělení.

```
1 import random
2
3 nahodne_cislo = random.normalvariate(0, 1)
4 print(nahodne_cislo)
```

Simulace hodu jednou koskou

```
1  import random
2
3  def hod_koskou():
4      return random.randint(1,6)
5
6  def simulace():
7      if (hod_koskou() == 1):
8          return True
9      else:
10         return False
11
12 POCETSIMULACI = 100000
13 uspesnych = 0
14 for i in range(POCETSIMULACI):
15     if (simulace()):
16         uspesnych += 1
17
18 print(uspesnych / POCETSIMULACI)
```

Narozeninový paradox

Narozeninový paradox jejev v teorii pravděpodobnosti, který říká, že v relativně malé skupině lidí je pravděpodobnost, že minimálně dva lidé sdílejí stejný narozeninový den, překvapivě vysoká. Konkrétně s pouhými 23 lidmi je šance 50%, tento počet je mnohem menší než intuitivní odhad.
Pro výpočet označme:

- d - počet dnů v roce
- n - počet lidí ve skupině

Pravděpodobnost, že všichni n jedinci mají různé narozeniny, je:

$$P(n) = \frac{d}{d} \times \frac{d-1}{d} \times \frac{d-2}{d} \times \ldots \times \frac{d-n+1}{d} = \frac{d!}{(d-n)! \times d^n}$$

Pravděpodobnost $\tilde{P}$, že minimálně dva lidé sdílejí narozeniny, je doplněk této pravděpodobnosti:

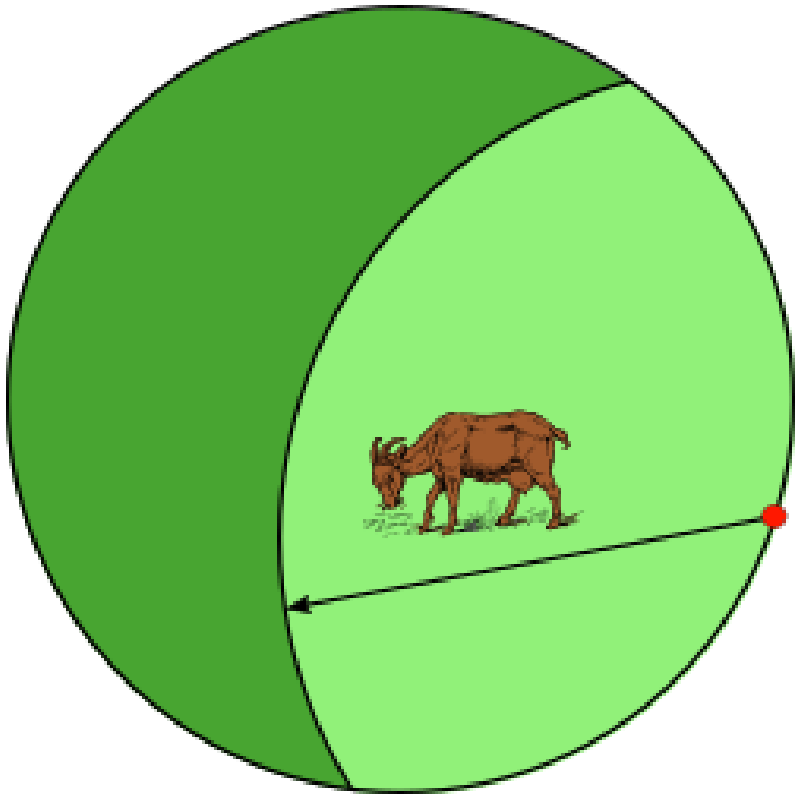
$$\tilde{P}(n) = 1 - P(n) = 1 - \frac{d!}{(d-n)! \times d^n}$$

Narozeninový paradox

```
1  import random
2  # Simulace narozeni - index 0 = 1.1, index 364 = konec roku
3  def simulace_narozeni():
4      return random.randint(0, 365)
5  def simulace(n):
6      narpole = [0 for x in range(366)]
7      for _ in range(n):
8          nar = simulace_narozeni()
9          narpole[nar] += 1
10         if (narpole[nar] > 1):
11             return True
12     return False
13 POCET = 10000
14 for i in range(1,365):
15     sum = 0
16     for _ in range(POCET):
17         sum += (1 if simulace(i) else 0)
18     pravd = sum/POCET
19     print(f"{i} - {pravd*100:.1f}%")
20     if (pravd >= 0.5): exit()
```

Problém pasoucí se kozy

Zahrada kruhového tvaru má poloměr r = 10m. Do zahrady umístíme kozu, kterou přivážeme provazem ke kolíku zatlučenému v obvodu zahrady. Jakou délku musí mít provaz, aby koza spásla trávu právě z poloviny plochy zahrady[5][6]? *Řešení je rovno* $r = 1.15872847\ldots$



[5] https://safehammad.com/post/2020/12/24/the-goat-of-monte-carlo/

[6] https://en.wikipedia.org/wiki/Goat_grazing_problem

Problém pasoucí se kozy - 1

```
1  import random
2
3  # Koza [0,0], kruznice [0,1]
4  def koza(r):
5      x = random.uniform(-1, 1)
6      y = random.uniform(-1, 1)
7      # Provadime dokud koza nebude v pozemku
8      while (x*x + y*y > 1):
9          x = random.uniform(-1, 1)
10         y = random.uniform(-1, 1)
11     # Jsme tam, kde koza muze jist?
12     return (x*x + (y+1)*(y+1) <= r*r)
13
14 def simulace(N, r):
15     sum = 0
16     for _ in range(N):
17         sum += (1 if koza(r) else 0)
18     print(f"{r} = {sum/N}")
19     return sum/N
```

Problém pasoucí se kozy - 2

```
20 N = 1_000_000
21 R = 1
22 d_mez = 0
23 h_mez = R*2 # odhadnuta inicialni hodnota
24
25 # Puleni intervalu
26 while (abs(d_mez - h_mez) > 1e-4):
27     n_mez = 0.5 * (d_mez + h_mez)
28     hodnota = simulace(N, n_mez)
29     if hodnota > 0.5:
30         h_mez = n_mez
31     elif hodnota < 0.5:
32         d_mez = n_mez
33     else:
34         break
35 print(f"Reseni je {0.5*(d_mez + h_mez)}")
36 # Exaktni reseni(pro r = 1):  1.15872...
```

Funkce `print()` a formátování výstupu

Úvod k funkci `print`

- Funkce `print()` v Pythonu slouží k zobrazení výstupu na obrazovce.
- Je to všestranná funkce, která umožňuje přizpůsobení formátu výstupu.
- Tisk může být použit ve funkcích k zobrazení mezivýsledků nebo finálních výstupů.
- Syntaxe:

```
1 print(value(s), sep=' ', end='**', file=soubor, flush=
      True)
```

  - **sep** slouží k určení oddělovače mezi hodnotami, které jsou vytisknuty. Ve výchozím nastavení je oddělovač mezerou
  - **end** určuje znak nebo řetězec, který je vytisknut na konci výstupu. Ve výchozím nastavení `end='\n'` přidá na konec výstupu nový řádek
  - **file** slouží k určení souboru, kam se výstup zapíše. Ve výchozím nastavení je výstup tisknut na standardní výstup (obvykle do konzole)
  - **flush** řídí, zda je výstupní buffer po tisku vyprázdněn. Pokud je `flush=True`, vyprazdňuje se výstupní buffer a všechna data budou okamžitě zapsána do souboru nebo konzole, jinak buffer nemusí být okamžitě vyprázdněn, což může zlepšit výkon ve vybraných případech.

f-strings v Pythonu

- f-strings jsou funkcí v Pythonu pro formátování řetězců tak, aby bylo možné zahrnout hodnoty proměnných způsobem, který je stručný a snadno čitelný.
- f-strings vytvoříte přidáním prefixu `f` nebo `F` k řetězci a zahrnutím složených závorek `{}` s výrazy uvnitř, které se mají vyhodnotit a formátovat.
- ve složených závorkách je možno použít i výraz, například `{cena * 100.0}`
- Příklad:

```
1     vyrobek = 'Tycinka musli'
2     cena = 80
3     print(f"Vyrobek {vyrobek} stoji {cena} Kc.")
4
```

Úvod do Pokročilého Formátování Řetězců v Pythonu

Pokročilé formátování řetězců v Pythonu poskytuje podrobnou kontrolu nad způsobem, jak jsou data prezentována, v souladu s specifikacemi uvedenými v PEP 3101. Umožňuje přesné formátování, včetně zarovnání, doplnění, správy znamének a dalších.

```
    num = 42
    formatted_num = f'Cislo: {num:+>10}'
    print(formatted_num)

```

```
    pi = 3.14159
    formatted_pi = f'Hodnota Pi: {pi:+10.2f}'
    print(formatted_pi)

```

Obecná Forma Standardních Formátovacích Specifikátorů

Obecná forma standardního formátovacího specifikátoru v Pythonu je

```
[[fill]align][sign][#][0][minwidth][.precision][type]
```

- `fill` - (nepovinný znak) definuje znak, který se použije pro vyplnění pole na minimální šířku. Za znakem výplně, pokud je přítomen, musí následovat příznak zarovnání.
- `align` - volitelný zarovnávací příkaz může být jedním z následujících:
  - < - zarovná hodnotu doleva uvnitř určené šířky.
  - > - zarovná hodnotu doprava uvnitř určené šířky.
  - ^ - zarovná hodnotu na střed uvnitř určené šířky.
  - = - nutí vložit mezery po znaménku (pokud existuje), ale před číslicemi.
  - ' ' - vloží mezeru před hodnotou, efektivně ji zarovná doprava a zajistí, že kladná čísla začínají vždy s mezerou.

Obecná Forma Standardních Formátovacích Specifikátorů

- `sign` - volitelný znaménkový příznak, platí pouze pro číselné typy a může mít jednu z následujících hodnot:
  - '+': Vynutí zahrnutí znaménka (+ nebo -) před číslicí.
  - '-': Pouze negativní čísla jsou předznačena znaménkem.
  - ' ': Kladná čísla jsou předznačena mezerou a negativní znaménkem.
- Pokud je přítomen znak #, používají celá čísla pro formátování *alternativní tvar*. To znamená, že binární, oktalový a hexadecimální výstup bude mít předponu `0b`, `0o` a `0x`.
- Pokud je před polem šířky uveden znak nula `0`, je povoleno doplňování nulou. To odpovídá typu zarovnání = a znaku výplně `0`.
- `minwidth` je celé číslo definující minimální šířku pole. Pokud není zadána, bude šířka pole určena obsahem.

Obecná Forma Standardních Formátovacích Specifikátorů

- `precision` je desetinné číslo udávající, kolik číslic se má zobrazit za desetinnou čárkou při převodu s plovoucí desetinnou čárkou. U nečíselných typů pole udává maximální velikost pole - jinými slovy, kolik znaků bude použito z obsahu pole. U celočíselných převodů se přesnost ignoruje.
- `type`
  - Volitelný typ příznaku může být jeden z následujících pro celá čísla:
    - `b`: Binární formát
    - `c`: Formát znaku
    - `d`: Desetinný formát
    - `o`: Osmičkový formát
    - `x`: Hexadecimální formát (malá písmena)
    - `X`: Hexadecimální formát (velká písmena)
    - `n`: Číselný formát (lokální specifický)
  - Pro čísla s pohyblivou řádovou čárkou jsou k dispozici následující typy
    - `e`: Exponenciální zápis (malé 'e')
    - `E`: Exponenciální zápis (velké 'E')
    - `f`: Fixní desetinný zápis
    - `g`: Obecný formát (pokud je exponent větší než -4 nebo menší než přesnost, použije se vědecký zápis, jinak desetinný zápis)

Ukázky použití f-strings

- f-string s výrazem

```
num1, num2 = 10, 20
print(f"The sum of {num1} and {num2} is {num1 + num2}.")
```

- f-string s použitím slovníků

```
person = {'name': 'John', 'age': 25}
print(f"Person's name is {person['name']} and age is {
    person['age']}.")
```

- f-string s formátovaným výstupem:

```
>>> cislo = 123.456789
>>>
>>> print(f"{cislo:X>+20}")
XXXXXXXXX+123.456789
>>> print(f"{cislo:X<20}")
123.456789XXXXXXXXXX
>>> print(f"{cislo:X<20.3f}")
123.457XXXXXXXXXXXX
>>> print(f"{cislo:X<20.3f}")
123.457XXXXXXXXXXXX
```

Moduly v Pythonu

Moduly v Pythonu

- Moduly v Pythonu jsou soubory obsahující kód v jazyce Python, který definuje funkce, třídy a proměnné, které lze použít v jiných programech v Pythonu.
- Pomáhají organizovat kód a usnadňují opakovatelnost kódu.
- Moduly jak ze standardní knihovny, tak z třetích stran
- Pro použití kódu z modulu v programu v Pythonu je třeba importovat modul pomocí příkazu `import`.
- Modul je možné importovat do dalšího modulu, či do *main* modulu (zdrojový soubor který byl spuštěn)
- Standardní knihovna Pythonu obsahuje širokou škálu modulů poskytujících funkce pro úkoly jako jsou operace se soubory, práce sítěmi, matematika a další.

```
1 import math
2
3 print(math.pi)
```

Moduly v Pythonu — ukázka

- Jméno souboru je jméno modulu s příponou `.py`. Například soubor `faktorial.py` definuje modul faktorial:

```
1  # Faktorial module
2
3  def faktorial(n):
4      """Funkce na vypocet faktorialu"""
5      sum = 1
6      for i in range(2, n + 1):
7          sum *= i
8      return sum
9
10 if (__name__ == "__main__"):
11     for i in range(10):
12         print(f"Factorial({i}) = {faktorial(i)}")
```

- Pokud budeme chtít pracovat s modulem `faktorial`, je potřeba modul importovat:

```
1  import faktorial
```

Moduly v Pythonu

- Modul může obsahovat spustitelné příkazy i definice funkcí. Tyto příkazy jsou určeny k inicializaci modulu. Jsou provedeny pouze při prvním setkání s názvem modulu v příkazu import
- Každý modul má svůj vlastní soukromý jmenný prostor, který je používán jako globální jmenný prostor všemi funkcemi definovanými v modulu.
- Moduly mohou importovat jiné moduly. Je obvyklé, ale ne povinné, umístit všechny příkazy import na začátek modulu (nebo skriptu, když na to přijde).
- Existuje varianta příkazu import, která importuje názvy z modulu přímo do jmenného prostoru importujícího modulu. Například:

```
1 from pocitani import soucet, soucet2
2 soucet(10,34.5)
```

Tím se nezavádí název modulu, z něhož jsou importy v lokálním jmenném prostoru převzaty (není definován modul `pocitani`).

Porozumění proměnné __name__ v modulech

- Proměnná `__name__` v modulech v Pythonu je speciální vestavěnou proměnnou, která uchovává název aktuálního modulu.
- Když je modul spuštěn jako hlavní program, `__name__` je nastaveno na `__main__`; když je importován jako modul, `__name__` je nastaveno na název modulu.

```
1 def soucet(c1, c2):
2     return c1 + c2
3
4 # Je spousten primo?
5 if __name__ == "__main__":
6     print("Tento modul je spusten primo.")
7 else:
8     print("Tento modul je importovan.")
```

Moduly v Pythonu

- Importování všech jmen, která modul definuje:

```
1 from pocitani import *
2 soucet(10,34.5)
```

  Tím se nezavádí název modulu, z něhož jsou importy v lokálním jmenném prostoru převzaty (není definován modul `pocitani`). Tento zápis se nedoporučuje,importují všechna jména kromě těch, která začínají podtržítkem _

- Pokud za názvem modulu následuje as, pak se jméno následující za as váže přímo na importovaný modul.

```
1 import pocitani as poc
2 poc.soucet(10,34.5)
```

Cesta k hledání modulů v Pythonu

- když je importován modul, interpretr hledá modul v built-in modulu stejného jména. Tyto moduly se dají vypsat pomocí proměnné `sys.builtin_module_names`
- pokud modul není nalezen je hledán soubor v listu adresářů které jsou definovány proměnnou `sys.path`. Tato proměnná je inicializována:
  - adresář který obsahuje zdrojový skript
  - adresáře v shell proměnné `PYTHONPATH`
  - ...
- funkce `dir()` vytiskne všechna jména aktuálně definovány, `dir(modul)` vytiskne všechna jména definována v daném modulu

Přehled Balíčků v Pythonu

- Balíčky v Pythonu jsou způsobem organizování souvisejících modulů do jednoho hierarchicky uspořádaného adresáře - tato hierarchická struktura umožňuje lepší organizaci kódu a snazší navigaci v související funkčnosti.
- Pomáhají modulovat kód, vyhnout se konfliktům v pojmenování a zlepšit organizaci a údržbu kódu.
- Pro vytvoření balíčku v Pythonu je potřeba vytvořit soubor `__init__.py` uvnitř adresáře.
- Soubor `__init__.py` může být prázdný, nebo může obsahovat iniciační kód balíčku.
- Moduly uvnitř balíčku lze importovat pomocí tečkové notace, kde se zadá nejprve název balíčku a poté název modulu.
- Například pro import modulu `body` z balíčku `cvut` se použije `import cvut.body`.

Jména a hodnoty

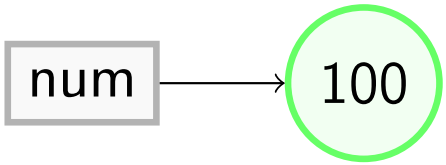
Mutable and immutable objekty

- Mutable objekty v Pythonu mohou být po vytvoření změněny.
- Immutable objekty v Pythonu nelze měnit po jejich vytvoření, zachovávají svůj původní stav.
- Příklady immutable objektů: celá čísla (`int`), reálná čísla (`float`), řetězce (`str`), n-tice (`tuple`), neměnná posloupnost bytů (`bytes`), bytové pole (`bytearray`), zmrazené množiny (`frozenset`), komplexní čísla (`complex`), boolovské hodnoty (`bool`), logický datový typ představující hodnoty `True` nebo `False`.
- Příklady mutable objektů: seznamy (`list`), slovníky (`dict`), množiny (`set`), pole bytů (`bytearray`), uživatelsky definované třídy (objekty vytvořené z uživatelsky definovaných tříd mohou být měnitelné podle toho, jak jsou implementovány).

Jména a hodnoty[7]

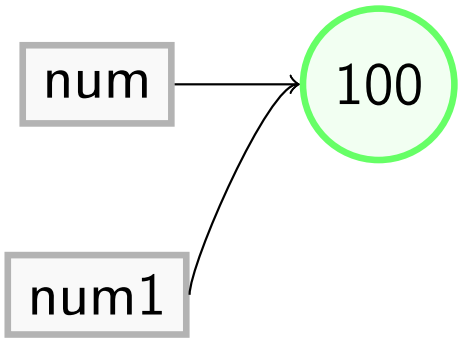
- **Přiřazení** zajistí že jméno odkazuje na hodnotu

```
1 num = 100
2 print(num)  # 100
```



- Více jmen může odkazovat na stejnou hodnotu

```
1 num = 100
2 num1 = num # num1 neodkazuje na num, ale na hodnotu 100
3 print(cislo1)  # 100
```

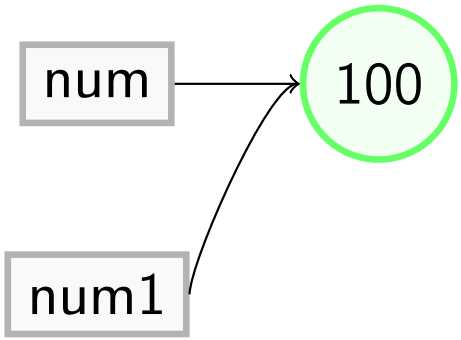


Převzato z `https://nedbatchelder.com/text/names1.html`

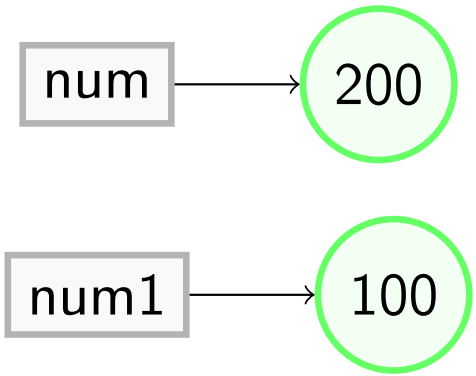
Jména a hodnoty ...[8]

- Jména jsou přiřazována nezávisle

```
1 num = 100
2 num1 = num
```




```
1 num = 200
```

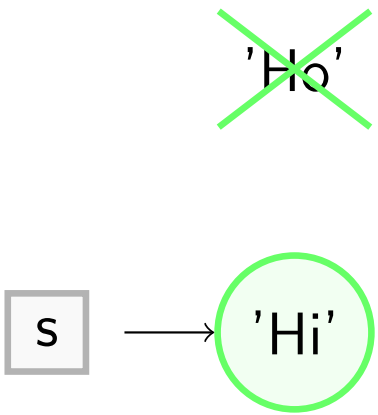


Převzato z https://nedbatchelder.com/text/names1.html

Jména a hodnoty ...[9]

- Hodnoty žijí dokud na ně je alespoň jeden odkaz (pak jsou smazány)

```
1 s = 'Ho'
2 s = 'Hi'
```



- **Přiřazení nikdy nekopírují data**

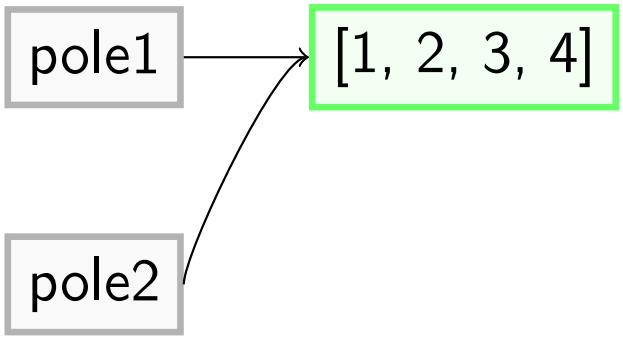
```
1 pole1 = [1, 2, 3]
2 pole2 = pole1
```

Převzato z https://nedbatchelder.com/text/names1.html

Jména a hodnoty ...[10]

- **Přiřazení nikdy nekopírují data**

```
1 pole1 = [1, 2, 3]
2 pole2 = pole1
3 pole1.append(4)
4 print(pole2) # [1, 2, 3, 4] !!!!!
```



- Toto neplatí pro immutable data

```
1 x = "hello"
2 y = x
3 x = x + " neco dalsiho"
4 # x = "hello neco dalsiho"
5 # y = "hello"
```

Převzato z https://nedbatchelder.com/text/names1.html

Jména a hodnoty ...[11]

- Varianty přiřazení (mutable a immutable objekty)

```
x += y
x = x + y           # koncepcne
x = x.__iadd__(y) # v podstate se vola __iadd__

# nasledujici dva radky jsou podobne
pole += [4, 5]
pole.extend([4, 5])
```

```
#Pseudo code
class List:
    def __iadd__(self, other):
        self.extend(other)
        return self
```

- Vše co se může vyskytnout na pravé straně přiřazení je reference (elementy listu, hodnoty a klíče slovníku, ...)

[11] Převzato z https://nedbatchelder.com/text/names1.html

Jména a hodnoty ...[12]

- Co vše je přiřazení:

```
x = ...
for x in ...
def x(...)
def fce(x):
...
```

- Příklad pro `for` cyklus

```
cisla = ["ahoj", "cau", "hello"]
for s in cisla:  # prirazeni s = cisla[0] ...
  s.upper()
print (cisla). # zadna zmena
```

- https://pythontutor.com/

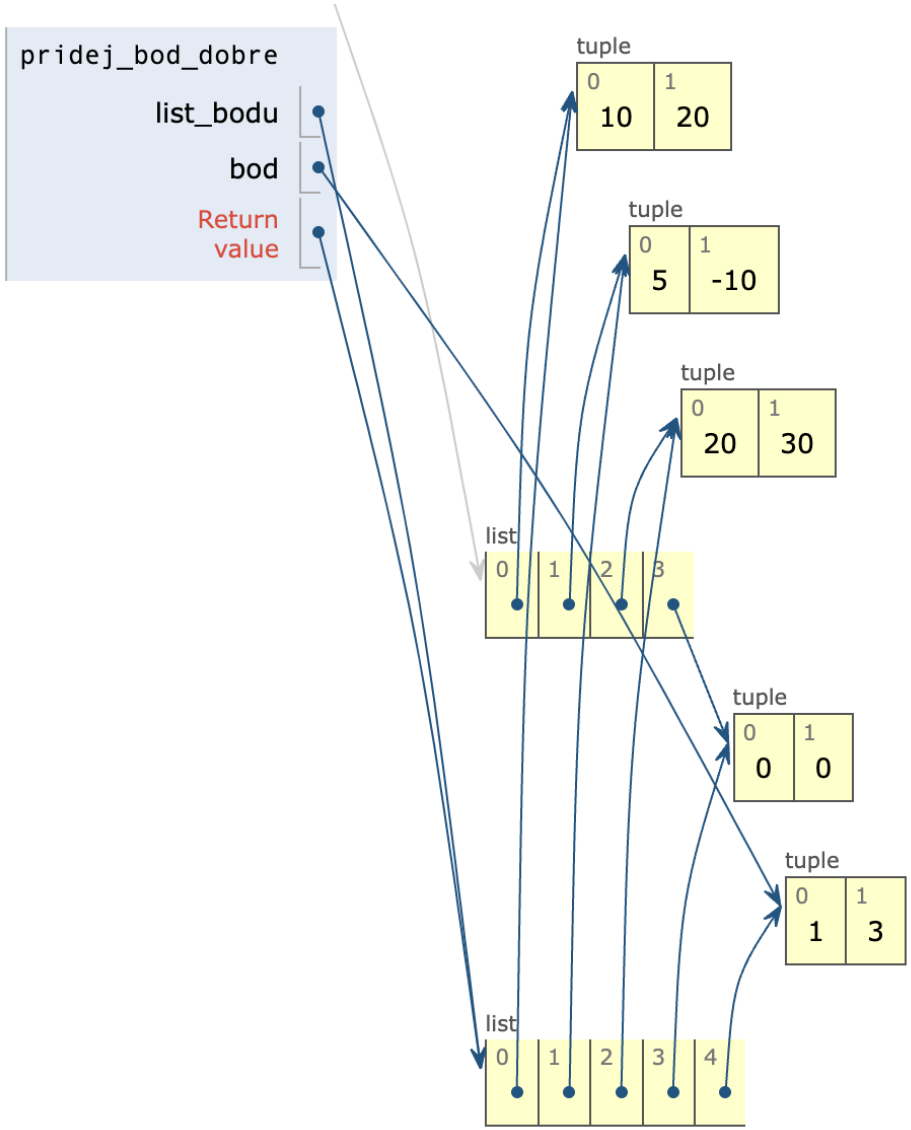
Převzato z https://nedbatchelder.com/text/names1.html

Předávání proměnných hodnotou a referencí

- Předávání proměnné hodnotou či předávání referencí - takto uvažovat v Pythonu je matoucí viz několik posledních slide.

```
1  def pridej_bod(list_bodu, bod):
2    """Meni argumenty"""
3    list_bodu.append(bod)
4
5  def pridej_bod_spatne(list_bodu, bod):
6    """Nic neudela, zbytecna funkce!"""
7    list_bodu = list_bodu + [bod]
8
9  def pridej_bod_dobre(list_bodu, bod):
10   """Vrati novy list bodu"""
11   list_bodu = list_bodu + [bod]
12   return list_bodu
13
14 body = [(10,20), (5, -10) , (20, 30)]
15 pridej_bod(body, (0, 0))
16 pridej_bod_spatne(body, (24, 4))
17 pridej_bod_dobre(body, (1, 3))
```

Předávání proměnných hodnotou a referencí - vizualizace po volání poslední funkce



Objektově orientované programování

Úvod do Tříd v Pythonu

- Python podporuje Objektově Orientované Programování (OPP).
- V Pythonu je všechno objekt a objekty jsou instancemi tříd.
- Třídy jsou předlohou pro vytváření objektů s atributy a metodami.
- Poskytují způsob modelování entit z reálného světa a zapouzdřování dat a funkcionality.
- Třídy jsou nástroji pro organizaci kódu, tvorbu opakovatelných komponent a modelování komplexních systémů.

```
1  class PrvniTrida:
2      """Trida PrvniTrida demonstruje vytvoreni prvni
       tridy"""
3      retezec = "Toto je ma prvni trida"
4
5      def print(self):
6          print(f"<<<{self.retezec}>>>")
7
8  # Vytvoreni objektu o a zavolani metody print
9  o = PrvniTrida()
10 o.print()
```

Definice třídy

- Definice třídy se může vyskytnout kdekoli, definici třídy je možno upravovat.

```
1  class PrvniTrida:
2      """Trida PrvniTrida demonstruje vytvoreni prvni
       tridy"""
3      pass
4
5
6  PrvniTrida.retezec = "Toto je ma prvni trida"
7
8  # Vytvoreni objektu o
9  o = PrvniTrida()
10 print(o.retezec)
```

Atributy v třídách

- Atributy v třídách jsou proměnné, které uchovávají data spojená s instancemi třídy.
- Mohou být definovány v rámci definice třídy a přistupuje se k nim pomocí tečkové notace.
- Máme dva typy atributů: instance atributy a class atributy.
- **Class atributy**
  - Třídní atributy jsou sdílené mezi všemi instancemi třídy.
  - Jsou definovány mimo jakoukoliv metodu v rámci třídy a přistupuje se k nim pomocí názvu třídy.
- **Instance atributy**
  - Instance atributy jsou specifické pro každou instanci třídy.
  - Jsou definovány v rámci metody `__init__` a přistupuje se k nim pomocí `self`.
- Přístup k atributům: `Trida.class_attribut`, `objekt.instance_atribut`

Atributy v třídách - ukázka

```
1  class Bod:
2      sourSystem = 'SJTSK' # sdilena vsemi instancemi
3      def __init__(self, name, x, y, sourSystem = "SJTSK"):
4          self.name = name
5          self.x = x
6          self.y = y
7          Bod.sourSystem = sourSystem     # unikatni atributy
       pro kazdou instanci
8      def __str__(self):
9          return f"Bod {self.name}[x,y][{Bod.sourSystem}] = [{
       self.x},{self.y}]"
10
11 bod1 = Bod("U studanky", 10,20)
12 print(bod1) # Bod U studanky[x,y][SJTSK] = [10,20]
13 bod2 = Bod("B12344", 100,300)
14 print(bod2) # Bod B12344[x,y][SJTSK] = [100,300]
15 bod3 = Bod("B3433", 11100,24304, "Neznamy")
16 print(bod3) # Bod B3433[x,y][Neznamy] = [11100,24304]
17 print(bod1) # Bod U studanky[x,y][Neznamy] = [10,20]
18 print(bod2) # Bod B12344[x,y][Neznamy] = [100,300]
```

Funkce v třídách

- Funkce v třídách jsou metody, které definují chování spojené s instancemi třídy.
- Jsou definovány v rámci třídy k provedení konkrétních akcí nebo operací.
- Funkce v třídě jsou definovány uvnitř třídy pomocí klíčového slova `def`.
- Musí mít `self` jako první parametr pro přístup k atributům a metodám.

```
1  class Student:
2      def __init__(self, jmeno, vek):
3          self.jmeno = jmeno
4          self.vek = vek
5
6      def tisk(self):
7          print(f"Student jmeno {self.jmeno}, vek {self.vek}")
8
9  #st = Student("Jan", 28)
10 #st.tisk()
```

Příklad Funkce v Třídě

```
1  class Student:
2      def __init__(self, jmeno, vek, predmety):
3          self.jmeno = jmeno
4          self.vek = vek
5          self.predmety = predmety
6
7      def pozdrav(self):
8          return f"Student {self.jmeno} ma zapsano {len(self.
       predmety)} predmetu"
9
10     def pridejPredmet(self, predmet):
11         self.predmety.append(predmet)
12
13 st = Student("Jan", 28, ["Fyzika", "Databaze"])
14 print(st.pozdrav())
15 st.pridejPredmet("Matematika")
16 print(st.pozdrav())
```

Konstruktory v Pythonu

- Konstruktory v třídách jsou speciální metody používané k inicializaci objektů při jejich vytváření.
- Metoda `__init__` slouží jako konstruktor, metoda která se volá při vytváření objektu.
- Přijímá parametr `self` a další parametry k inicializaci atributů objektu – umožňuje nastavit počáteční hodnoty pro vlastnosti objektu.
- Není možné mít pro danou třídu definováno více konstruktorů.

```
1  class Student:
2      def __init__(self, jmeno, vek = 99):
3          self.jmeno = jmeno
4          self.vek = vek
5
6      def tisk(self):
7          print(f"Student jmeno {self.jmeno}, vek {self.vek}")
8  st = Student("Jan", 28)
9  st.tisk() # Student jmeno Jan, vek 28
10 st1 = Student("Karel")
11 st1.tisk() # Student jmeno Karel, vek 99
```

Vytváření Objektů z Tříd

- Pro vytvoření(inicializaci) objektu z třídy použijte název třídy následovaný závorkami. Tím se zavolá konstruktor třídy k inicializaci objektu.
- Jakmile je objekt vytvořen, můžete přistupovat k jeho atributům a metodám pomocí tečkové notace.

```
1  class Trida:
2      pocet = 0
3      def __init__(self, cislo):
4          self.cislo = cislo
5          Trida.pocet += 1
6      def getPocet(self):
7          return Trida.pocet
8      def getCislo(self):
9          return self.cislo
10 t1 = Trida(-100)
11 print(f"p = {t1.getPocet()}, c = {t1.getCislo()}")
12 for i in range(0,100):
13     t2 = Trida(i*10)
14     print(f"p = {t2.getPocet()}, c = {t2.getCislo()}")
```

Privátní Proměnné v Python Třídách

- Python podporuje skrytí dat pomocí použití privátních atributů a metod s pomocí konvence podtržítka.
- Privátní proměnné v Python třídách jsou proměnné, ke kterým by mělo být přistupováno pouze uvnitř třídy.
- Privátní proměnné pomáhají při zapouzdření a skrytí dat, což umožňuje udržet vnitřnosti třídy skryté před vnějším zásahem.
- Jsou označeny zdvojeným podtržítkem před názvem proměnné __.

```
1  class Tajne:
2      def __init__(self, sifra):
3          __sifra = sifra
4      def __vypis_sifru(self):
5          print(f"Sifra = {self.__sifra}")
6  tajne = Tajne("ABCDEFGH")
7  tajne.__vypis_sifru()
8  # AttributeError: 'Tajne' object has no attribute '
       __vypis_sifru'
9  print(tajne.__sifra)
10 # AttributeError: 'Tajne' object has no attribute '
       __vypis_sifru'
```

Výchozí metody v třídách

- Výchozí metody v Pythoních třídách jsou speciální metody, které poskytují výchozí chování pro objekty.
- Tyto metody jsou volány v konkrétních situacích a mohou být upraveny v uživatelsky definovaných třídách.
  - **Metoda** `__str__` - slouží k definování tisknutelného řetězcového zobrazení objektu. Je volána při použití funkce `str()` na objekt nebo když je předán funkci `print()`.
  - **Metoda** `__eq__` - slouží k definování chování rovnosti operátoru `==` pro objekty. Porovnává dva objekty a vrátí `True`, pokud jsou považovány za rovné na základě vlastní implementace.