Python

Pokročilé programování

Petr Gregor

ICT Pro

15.-17. prosince 2025

Petr Gregor

- Lektor
  - Python
  - Java
  - AI
  - Data Science
- Analytik na MUNI
- Programátor (Pascal, C, C++, C#, Java, JavaScript, PHP, VB, Python, Ruby,...)
- 25+ let zkušeností s výukou
- 20+ let zkušeností s programováním

ictPRO

Obsah školení: Část 1/4

1. Přehled základních vlastností jazyka
   - Proměnné a reference
   - Standardní datové typy
   - Řídící struktury
2. Organizace projektu
   - Funkce
   - Třídy a metody
   - Moduly a balíčky
   - Distribuce software
3. Funkce a metody
   - Poziční a pojmenované parametry
   - Pokročilé zpracování argumentů
   - Vnořené funkce a funkcionální prvky
   - Globální, lokální a vázané proměnné
   - Statická analýza kódu

Obsah školení: Část 2/4

4. Objektový model
   - Objektově orientovaný návrh
   - Datové typy a operace
   - Speciální metody
   - Deskriptory
   - Dědičnost a polymorfie
   - Vícenásobná dědičnost a mixins
   - Základní návrhové vzory
5. Generátory a iterátory
   - Sekvenční datové typy
   - Generované sekvence
   - Čtení textových souborů
6. Ukládání a zpracování dat
   - Ukládání objektů
   - Přístup k databázi

Obsah školení: Část 3/4

7. Ladění a logování
   - Ladění pomocí výpisů
   - Standardní logovací knihovna
   - Debuggery
8. Online komunikace
   - Webový klient a server
   - Práce s API
9. Testování
   - Testování knihoven a aplikací
   - Organizace testů

Obsah školení: Část 4/4

10. Paralelní zpracování
    - Vlákna a procesy
    - Komunikace a synchronizace
    - Výkonnostní omezení Pythonu

11. Pokročilá témata
    - Generování kódu
    - Monkey patching

Přehled základních vlastností jazyka

Proměnné

Proměnné jsou kontejnery pro ukládání dat. Představte si proměnné jako krabice, kam můžeme něco uložit, např. hodnotu (implicitně).
Definice proměnné v Pythonu:

```
1 number = 10
2 word = "text"
3 big_number = 99999.999
4 truth = True
```

Proměnné

Proměnná:

- má název,
- má hodnotu,
- má své místo v paměti počítače.

Konvence pojmenování proměnných

- Hodnotu proměnné můžete změnit přiřazením nové hodnoty.
- Název proměnné může obsahovat pouze písmena: `a-z`, `A-Z`, čísla `0-9` a symbol `_`.
- Je **citlivý na velikost písmen (case sensitive)**!
- Název proměnné **nesmí začínat číslicí**!
- V Pythonu se proměnné pojmenovávají ve stylu:
  - **snake case** (hadí styl): každé slovo je odděleno podtržítkem (`very_big_number`)
  - **CamelCase** (velbloudí styl): nové slovo začíná velkým písmenem (`VeryLongString`) – tento styl je rezervován pro třídy.

Konvence pojmenování proměnných

Povolená řešení:

```
SUPER_VARIABLE = 1
bestVariableEver123 = "OK"
_my_name = "Alice"
another1_variable2 = 5.5
_____ = True
```

Nepovoleno:

```
123variable = 2
wrong$name%123 = False
!@AlmOStGoODone!@ = None
12345 = "bad_name"
VARIABLE-1 = 0
```

Reference

V Pythonu proměnné neukládají přímo hodnoty (jako např. v C++ nebo starším Pascalu). Místo toho fungují jako **nálepky** nebo **názvy**, které **referují** na objekt uložený v paměti.

- **Objekt v paměti:** Skutečná hodnota (např. číslo `10`, řetězec `'Hello'`, seznam `[1, 2]`) je fyzicky uložena jako **objekt** v paměti.
- **Proměnná jako reference:** Když napíšete `x = 10`, proměnná `x` se stane referencí ukazující na objekt s hodnotou `10`.
- **Sdílené reference:** Když napíšete `y = x`, proměnná `y` **nekopíruje** hodnotu. Místo toho se stane **druhou referencí** ukazující na **tentýž objekt** v paměti jako `x`.

Reference

Identifikace objektů (`id()` a `is`)

- `id(objekt)`: Vestavěná funkce, která vrací jedinečné číslo identifikující objekt v paměti. Pokud mají dvě proměnné stejné `id()`, odkazují na tentýž objekt.
- `is` **operátor:** Kontroluje, zda dvě reference ukazují na **stejný objekt** (kontroluje `id`), na rozdíl od `==`, který kontroluje, zda jsou si objekty **hodnotově rovny**.

Měnitelnost (*mutable*) vs. neměnitelnost (*immutable*)

Chování referencí je nejlépe vidět při rozlišování mezi **neměnnými (immutable)** a **měnitelnými (mutable)** datovými typy.

Neměnné

Neměnné (Integers, Strings, Tuples) — Pokud se pokusíte „změnit“ hodnotu neměnného objektu, Python **vytvoří nový objekt** v paměti, a proměnná se na něj přesměruje.

```
1  # Immutable data type: int
2  a = 10
3  b = a
4
5  print(f"Before change: id(a) = {id(a)}, id(b) = {id(b)}")
6  print(f"a is b: {a is b}")  # True
7
8  # Changing the value of 'a'
9  a = 20 # Python creates a NEW object (20) and redirects 'a'
       to it.
10 print(f"After change: id(a) = {id(a)}, id(b) = {id(b)}")
11 print(f"a is b: {a is b}")  # False (a and b no longer point
       to the same location)
12 print(f"b remains: {b}")    # b is still 10
```

Měnitelné

Pokud modifikujete měnitelný objekt, Python **nemění referenci**; místo toho změní **obsah objektu v paměti**, na který ukazují **všechny** reference.

```
# Mutable data type: list
list_a = [1, 2, 3]
list_b = list_a # list_b is reference to the SAME object

print(f"Before change: id(list_a) = {id(list_a)}, id(list_b) = {id(list_b)}")
print(f"list_a is list_b: {list_a is list_b}") # True

# Modifying the object via list_a
list_a.append(4) # Changes the content of the object

print(f"After changing list_a: id(list_a) = {id(list_a)}, id(list_b) = {id(list_b)}")
print(f"list_a is list_b: {list_a is list_b}")  # Still True (They still point to the same location)
print(f"list_b also changed: {list_b}")  # list_b is now also [1, 2, 3, 4]
```

Typy

Python vám umožňuje specifikovat typ proměnné. Toto je volitelné a je pouze návrhem.

```
1 number: int = 5
```

Standardní datové typy

Skutečnost, že typy v Pythonu jsou pouze návrhem, způsobuje, že následující kód funguje bez problémů:

```
1 number: int = 'i am a string'
2
3 print(type(number))  # prints <class 'str'>
```

Tisk proměnných na obrazovku

Proměnné ukládají hodnoty, takže je lze použít jako parametry funkcí.

```
1 number = 10
2 new_word = "new string"
3
4 print(number)
5 print(new_word)
6
7 print(type(number))
8 print(type(new_word))
```

Standardní datové typy

Základní datové typy v Pythonu:

- `int` – celá čísla (0, 1, -3, 128, ...).
- `float` – čísla s plovoucí desetinnou čárkou (reálná čísla: 0.123, 256.2, -3.14).
- `complex` – komplexní čísla ((1-2j), (30+15j)).
- `str` a `bytes` – textové sekvence ("text", 'single quotes', b"bytes sequence").
- `bool` – logické datové typy true/false (True, False).
- `NoneType` – speciální, **nedefinovaný** typ neexistujících hodnot (None).

Tisk různých typů dat na obrazovku

```
1 print("Text to print.")
2 print(-17)
3 print(123.4)
4 print(False)
5 print(None)
```

Testování typu proměnné

Pro práci s typy proměnných nabízí Python dvě hlavní funkce:

- `type(obj)`: Vrací typ (třídu) daného objektu. Používá se pro zjištění, co je proměnná zač.
- `isinstance(obj, type)`: Kontroluje, zda je objekt instancí daného typu (nebo jeho podtřídy). Vrací `True` nebo `False`.

Kontrola datového typu – funkce `type()`

V Pythonu můžeme zkontrolovat datový typ pomocí vestavěné funkce `type()`:

```
1 type("What is the data type of this data?")
```

- Funkce `type()` přebírá parametr stejným způsobem jako funkce `print()`.
- Funkce `type()` vrací textovou hodnotu, která je názvem typu jejího parametru.
- Datový typ můžeme vypsat kombinací obou funkcí:

```
1 print(type("What is the data type of this data?"))
2 print(type(10.2))
3 print(type(True))
```

Testování typu proměnné

`isinstance(obj, type)`: Kontroluje, zda je objekt instancí daného typu (nebo jeho podtřídy). Vrací `True` nebo `False`.

```
value = 42

# 1. Inspection
print(type(value))  # Output: <class 'int'>

# 2. Type Testing
# Recommended way for conditions
if isinstance(value, int):
    print("Value is an integer")
```

Řízení programu

Python podporuje instrukce, které mohou měnit pořadí, v jakém je kód vykonáván. Tyto instrukce jsou:

- **podmíněné příkazy** (`if`, `elif`, `else`, `match`),
- **cykly** (`for`, `while`).

Podmíněné příkazy

V reálném světě musíme často dělat nějaká rozhodnutí. **Jestliže** prší, **pak** si vezmu deštník. V programování nám příkaz `if` umožňuje dělat různá rozhodnutí v kódu, v závislosti na dané podmínce.

Příkaz `if`

Příklad jednoduchého použití příkazu `if`:

```
1 if <condition>:
2     <instructions>
```

- Hodnota `condition` musí být přeložitelná na logickou hodnotu (`True`/`False`).
- Pokud je `condition` pravdivá (přeloží se na `True`), Python vykoná `instructions`.
- Je vyžadováno odsazení!

Příkaz if – příklad

```
1 x = 0
2 y = 3
3
4 if x > y:  # This will be translated to False because 0 is
      not greater than 3
5     print(f"{x} is greater than {y}")  # This will not be
          displayed
6
7 if x < y:  # This will be translated to True because 3 is
      greater than 0
8     print(f"{x} is less than {y}")  # This will be displayed
```

Odsazení

- Rozpoznávací znak Pythonu.
- Aby bylo možné vykonat více než jednu instrukci v bloku `if`, musí být všechny instrukce v kódu odsazeny.

```
if condition:
    instruction_1
    instruction_2
    instruction_3
    ...
    instruction_n
Next_instructions_after_if_block
```

Odsazení se v Pythonu používá k vytváření bloků kódu nebo složených příkazů.

Příkaz else

Stejně jako v reálném životě, programování umožňuje provést jinou volbu, pokud není splněna určitá podmínka. K tomu se používá klauzule `else`.

```
1 if condition:
2     instructions
3 else:
4     other_instructions
```

Příkaz `else` je volitelný a po příkazu `if` může následovat nanejvýš jeden příkaz `else`.

Příkaz else – příklad

```
1 x = 0
2 y = 3
3
4 if x > y:  # This will be translated to False because 0 is
      not greater than 3
5     print(f"{x} is greater than {y}")  # This will not be
          displayed
6 else:  # This will be translated to True because 3 is greater
      than 0
7     print(f"{x} is less than {y}")  # This will be displayed
```

Příkaz elif

Existuje také způsob, jak provést jednu z mnoha voleb v závislosti na tom, která z dostupných podmínek bude splněna jako první.

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif condition3:
    instructions3
else:
    instructions4
```

Příkaz `elif`

- V podmíněném příkazu lze implementovat libovolný počet klauzulí `elif`.
- Klauzule `elif` je volitelná.
- Pokud není splněna žádná podmínka (ani pro `if`, ani pro žádný z příkazů `elif`), budou následovat instrukce v bloku `else` (pokud je přidán).

Příkaz elif

```
1 x = 0
2 y = 3
3
4 if x > y:  # This will be translated to False because 0 is
      not greater than 3
5     print(f"{x} is greater than {y}")  # It will not be
          displayed
6 elif x == 3:  # This will be translated to False because 0 is
      not equal to 3
7     print(f"{x} is equal {y}")
8 else:  # This will be translated to True because 3 is greater
      than 0
9     print(f"{x} is less than {y}")  # It will be displayed
```

Příkaz `match` (Python 3.10+)

- **Structural Pattern Matching:** Konstrukce zavedená v Pythonu 3.10.
- Funguje podobně jako `switch` v jiných jazycích, ale je mnohem mocnější (umožňuje rozbalování struktur).
- `case _`: Slouží jako výchozí větev (wildcard), pokud žádný jiný vzor neodpovídá.

```
1  status = 404
2
3  match status:
4      case 200:
5          print("Success")
6      case 400 | 404:
7          print("Client Error")
8      case 500:
9          print("Server Error")
10     case _:
11         print("Unknown status")
```

Iterace

Iterace je opakované provádění sady příkazů. Programové struktury, které implementují iterace, se nazývají **cykly**. V **nekonečné iteraci** není počet provedení cyklu předem určen. Daný blok kódu se provádí mnohokrát, dokud je splněna určitá podmínka. V **definované iteraci** se blok kódu zopakuje stanovený počet opakování.

Cyklus while

- Cyklus se provádí, dokud je `condition` pravdivá.
- Zkontroluje se, zda je hodnota `condition` `True`. Pokud ano, `instructions` se vykonají. Pokud ne - Python přeskočí blok cyklu a provede příkazy mimo něj.
- Po vykonání bloku cyklu `while` se `condition` znovu zkontroluje. Pokud je stále pravdivá, cyklus se provede znovu.

```
1 while condition:
2     <instruction_1>
3     <instruction_2>
4     ...
5     <instruction_n>
```

Cyklus while – příklad

Tento program vypíše čísla 1, 2, 3, 4, 5 - každé na nový řádek.

```
1 # Make loops as long as n is less than 5
2 n = 0
3 while n < 5:
4     n += 1 # increment n with each loop loop
5     print(n)
```

Ukončení cyklu

Příkaz `break`:

- Okamžitě zastaví aktuální iteraci a samotný cyklus.
- Program opustí blok cyklu a pokračuje v provádění instrukcí mimo něj.

Příkaz `continue`:

- Okamžitě zastaví aktuální iteraci a pokračuje další.
- Před zahájením dalšího cyklu se `condition` znovu zkontroluje. To určí, zda má k dalšímu cyklu dojít, nebo ne.

Cyklus while – příklad 2

Tento program vypíše čísla 2 a 3, každé na nový řádek.

```
# Make loops as long as n is less than 5
n = 0
while n < 5:
    n += 1 # increment n with each loop loop
    if n == 4: # if n is 4, end the loop
        break
    if n == 1: # if n is 1, start a new iteration
        continue
    print(n)
```

Cyklus for

```
1 for var in iterable:
2     instructions
```

- iterable je kolekce proměnných/hodnot, přes kterou můžeme iterovat – například seznam.
- K vytvoření bloku cyklu bude zapotřebí odsazení.
- Pomocí cyklu for můžeme vykonat sadu příkazů jednou pro každou položku v daném: list, tuple, set atd.
- Proměnná var přebírá hodnotu každého prvku v kolekci iterable a je k dispozici v cyklu.

Cyklus `for` – příklad

Program vytiskne všechny položky v seznamu.

```
1 animals = ["Dog", "Cat", "Fish"]
2
3 # List all animals from the animals list
4 for animal in animals:
5     print(animal) # Lists one animal in turn
```

Ukončení cyklu – `for`

Příkazy `break` a `continue` jsou plně podporovány také v cyklu `for`.

Funkce `range()`

`range(start, stop, step)`

- Funkce `range()` vrací iterovatelný objekt obsahující čísla od `0` do `start`, pokud je jako argument zadáno pouze číslo `start`.
- Funkce `range()` vrací iterovatelný objekt obsahující čísla od `start` do `stop` **s vyloučením** čísla `stop`, pokud jsou zadány `start` i `stop`.
- Volitelně můžete zahrnout parametr `step` určující, kolik prvků mezi hodnotami by mělo být přeskočeno.

Funkce range() – příklady

```
# Will print 0, 1, 2 in new lines
for i in range(3):
    print(i)

# Will print -3, -2, -1, 0 in new lines
for i in range(-3, 1):
    print(i)
```

```
# Will print 3, 5, 7, 9 in new lines
for number in range(3, 11, 2):
    print(number)

# Will print -1, -2, -3 in new lines
for number in range(-1, -4, -1):
    print(number)
```

Funkce `enumerate()`

- Často chceme při práci s iterátory také znát aktuální počet iterací.
- Funkce `enumerate()` přijímá kolekci jako parametr a vrací n-tici se dvěma hodnotami: indexem prvku a aktuálně uvažovaným prvkem.

```
1 fruits = ["apple", "banana", "lemon"]
2
3 for index, fruit in enumerate(fruits):
4     print(f"Fruit: {fruit}, under the index: {index}.")
```

List comprehension

- Představte si situaci, kdy chceme vytvořit seznam tisíce čísel od 0 do 999.
- Seznam je příliš velký pro ruční zadání hodnot.
- Může být naplněn hodnotami pomocí cyklu `for` nebo vytvořen pomocí mechanismu **list comprehension**.

List comprehension – příklad

```python
# List in loop for
numbers = []
for i in range(1000):
    numbers.append(i)

print(len(numbers)) # Prints 1000

# Folded list
numbers = [i for i in range (1000)]
print(len(numbers)) # Prints 1000
```

Dict comprehension

Podobně můžete použít mechanismus složení slovníku pro inicializaci slovníku.

```
1 keys_and_values = [(1, 'a'), (2, 'b'), (3, 'c')]
2 dictionary = {number: letter for (number, letter) in
      keys_and_values}
```

Organizace projektu

Funkce

- Pro zajištění principu *Neopakujte se* (DRY – *Don't Repeat Yourself*), by měl být jakýkoli blok kódu, který provádí opakovanou akci nebo výpočet, umístěn do funkce nebo metody.
- Funkce **zvyšují čitelnost**, snižují nároky na údržbu a dělají **kód snadnějším k ladění**.
- Funkce je blok znovupoužitelného kódu navržený k provedení jediného, specifického úkolu.
- Definuje se pomocí klíčového slova `def`.
- Funkce může přijímat parametry (viz později) jako vstup.
- Počítá a generuje výsledek na základě zadaných argumentů.
- Musí být definována předtím, než je poprvé použita v kódu.

Funkce

Funkce v Pythonu se definují zadáním klíčového slova `def`, názvu funkce, jejích možných parametrů v závorkách a napsáním nezbytných instrukcí v jejím bloku (nezapomeňte na odsazení!).

```
1 def function_name_1():
2     instructions
3
4 def function_name_2(arg_1, arg_2, ..., arg_n):
5     instructions
```

Funkce – příklad 1

```
1 # Definition of the function named print_hello_world
2 def print_hello_world():
3     print("Hello world from inside the function!")
4
5 # Calling print_hello_world()
6 print_hello_world()
```

Funkce – příklad 2

```
1 # Function definition of greet_by_name (name)
2 def greet_by_name(name):
3     print(f"Hello, {name}")
4
5 # Call function greet_by_name (name) with "John" as the name
      argument
6 greet_by_name("John")
```

Funkce – Příklad: Bez použití funkce (Non-DRY)

```
1  VAT_RATE = 0.21
2
3  # ─── Product A ───
4  price_a = 150.00
5  # Calculation logic starts here
6  vat_amount_a = price_a * VAT_RATE
7  total_price_a = price_a + vat_amount_a
8  # Calculation logic ends here
9
10 print(f"Product A:")
11 print(f"Base: {price_a:.2f}")
12 print(f"VAT:  {vat_amount_a:.2f}")
13 print(f"Total: {total_price_a:.2f}")
14
15 print("-" * 20)
16
17 # ─── Product B ───
18 price_b = 499.50
19 # The EXACT SAME calculation logic is repeated
20 vat_amount_b = price_b * VAT_RATE
```

Příklad: Použití funkce (DRY)

```python
VAT_RATE = 0.21

def calculate_total_price(base_price):
    """
    Calculates the VAT amount and the total price including VAT.
    Returns: (vat_amount, total_price)
    """
    vat_amount = base_price * VAT_RATE
    total_price = base_price + vat_amount
    return vat_amount, total_price

def display_result(product_name, base_price, vat_amount, total_price):
    """
    Formats and prints the results for a single product.
    """
    print(f"{product_name}:")
    print(f"Base: {base_price:.2f}")
    print(f"VAT: {vat_amount:.2f}")
```

Třídy a metody

- V Pythonu je všechno *objekt* (čísla, řetězce, funkce, třídy).
- Třídy jsou šablony (blueprints) pro vytváření objektů (instancí).
  - Definují se pomocí klíčového slova `class`.
  - Více později **??**.
- Metody jsou funkce definované uvnitř třídy.
  - Vždy přijímají referenci na instanci jako první argument (konvenčně `self`).
  - Více později **??**.

Moduly

- Modul je jediný soubor `.py`.
- Cíl – seskupování a znovupoužitelnost funkcí, tříd a proměnných.
- Každý modul má svůj vlastní jmenný prostor.

Moduly

POZNÁMKA:

Použijte blok `if __name__ == "__main__":` pro spuštění kódu pouze v případě, že je soubor spuštěn přímo.

Soubor `module_hello.py`:

```
1 name = "Jack"
2
3 def print_hello(name):
4     print(f"Hello, {name}!")
5
6 if __name__ == "__main__":
7     print_hello("Peter")
```

Importování modulů – první metoda

```
1 import module_hello
2
3 print(module_hello.hello)
4
5 module_hello.print_hello("Adam")
```

Importování modulů – druhá metoda

```
1 import module_hello as mh
2
3 print(mh.hello)
4
5 mh.print_hello("Black")
```

Importování modulů – třetí metoda

```
1 from module_hello import print_hello
2
3 # print(hello)  # ERROR
4
5 print_hello("Black")
```

Importování modulů – čtvrtá metoda

```
1 from module_hello import *
2
3 print(hello)  # OK
4
5 print_hello("Black")
```

Jak importovat správným způsobem?

- Použijte pouze jeden příkaz `import` na řádek!
- Nejlepší praxí je neimportovat třídy, proměnné atd. – pouze celé moduly.
- Importy jsou rozděleny do tří sekcí:
  - moduly standardní knihovny,
  - moduly externí knihovny,
  - moduly uvnitř projektu (náš kód).
- V rámci jedné sekce by importy měly být uvedeny v abecedním pořadí.

Jak Python najde moduly?

Při importování kódu Python hledá moduly počínaje od:

1. Aktuálního adresáře.
2. Cesty odvozené ze systémové proměnné `PYTHONPATH`.
3. Umístění standardních knihoven Pythonu.
4. Umístění externích knihoven Pythonu.

Jak Python hledá importované moduly?

Seznam cest, které budou prohledávány k nalezení importovaných modulů, lze získat z proměnné `path` v modulu `sys`:

```
1 import sys
2
3 print(sys.path)
```

Atribut `__file__`

Vestavěný atribut modulu `__file__` obsahuje **cestu k souboru**, ze kterého byl aktuální modul nebo skript načten.

- Je nezbytný pro konstrukci **absolutních cest** k jiným souborům (jako jsou konfigurační soubory, databáze nebo statické prostředky), které jsou relativní k umístění spouštěného skriptu, bez ohledu na aktuální pracovní adresář.
- Je automaticky dostupný ve všech **Python modulech**, které jsou importovány nebo spuštěny jako skript.
- Režim spuštění:
  - Pokud je soubor spuštěn přímo (např. `python my_script.py`), `__file__` bude obsahovat cestu použitou k jeho spuštění (často relativní cesta).
  - Pokud je soubor importován jako modul, `__file__` obsahuje jeho umístění v systému souborů.

Balíčky

- Balíček Pythonu je adresář, který obsahuje moduly a/nebo podbalíčky.
- Cíl – hierarchická organizace souvisejícího kódu.
- Balíček musí obsahovat speciální soubor s názvem `__init__.py`.
- Soubor `__init__.py` může být prázdný nebo obsahovat obecný kód pro balíček.

Jak importovat celé balíčky?

- Pokud je modul v umístění známém Pythonu, můžete jej importovat.
- Můžete například importovat jeden modul z balíčku.
- Můžete použít operátor * a importovat všechny moduly z balíčku.

```
1 # The path to the school package MUST be previously
2 # added to the PYTHONPATH env variable
3 import school
4 from school import student
5 from school import student, teacher
6 from school import *
7 from school import classroom as room
```

Nástroj PIP

- **P**ackage **I**nstaller for **P**ython.
- Balíček pro instalaci externích knihoven Pythonu.
- Vývojáři Pythonu vytvořili obrovské množství balíčků, které lze stáhnout pomocí nástroje `pip`.

Instalace závislostí

- Ne všechny užitečné nástroje jsou zahrnuty ve standardní knihovně a jsou distribuovány s Pythonem.
- Mnoho externích knihoven čeká na volitelnou instalaci (pouze v případě potřeby).
- `pip` se používá pro instalaci a správu dalších balíčků.
- Základní příkazy nástroje `pip` lze získat pomocí příkazu: `pip help`.
- Požadovaný balíček můžeme nainstalovat pomocí příkazu:
  `pip install <package_name>`
- Balíčky lze odinstalovat voláním:
  `pip uninstall <package_name>`

Nainstalované závislosti

- Chcete-li zkontrolovat všechny aktuálně nainstalované balíčky a jejich verze, použijte příkaz: `pip freeze`
- Chcete-li vyhledat nainstalované balíčky, které mají novější verzi ke stažení, použijte příkaz: `pip list -o`
- Zvolený balíček lze aktualizovat pomocí příkazu:
  `pip install -U <package_name>`

POZOR:

Program bude vždy používat jednu konkrétní verzi každé knihovny!

Nainstalované závislosti

- Knihovny lze nainstalovat s ohledem na konkrétní verzi pomocí příkazu: `pip install <package_name>==version`
- Pokud nechceme instalovat konkrétní verzi balíčku (protože může obsahovat chyby), můžeme si jej stáhnout v jiné verzi, než jsme původně uvedli, pomocí následujícího příkazu:
  `pip install <package_name>!=version`
- Instalace knihovny pouze od vybrané verze:
  `pip install <package_name> >=version`
- Instalace balíčku mezi verzemi:
  `pip install <package_name>>=version_min, <version_max`

Externí knihovny v projektech

- V pokročilých projektech se používá mnoho dalších balíčků.
- Pokud na projektu pracuje několik programátorů, měli by používat přesně stejnou verzi knihovny.
- Je dobrým zvykem ukládat názvy všech závislostí spolu s jejich přesnými verzemi do formátu kompatibilního s nástrojem `pip`.
- Pro vytvoření souboru *requirements.txt*, který ukládá informace o nainstalovaných externích knihovnách, použijte příkaz:
  ```
  pip freeze > requirements.txt
  ```
- Pro obnovení prostředí (obnovením nainstalovaných knihoven) ze souboru *requirements.txt* spusťte příkaz:
  ```
  pip install -r requirements.txt
  ```

Virtuální prostředí

- Na svém počítači můžete vytvořit mnoho různých projektů.
- Každý z nich může používat stejné knihovny, ale v různých, nekompatibilních verzích.
- Znamená to, že musíme mít různé projekty na různých počítačích?

Virtuální prostředí

- Cíl: Izolovat závislosti pro každý projekt, aby se zabránilo konfliktům verzí mezi projekty.
- Nástroje: `venv` (vestavěný), `conda`, nebo `poetry`.
- Správa závislostí: Soubor *requirements.txt* uvádí externí balíčky k instalaci přes `pip`.

Co je PyPI?

- **Py**thon **P**ackage **I**ndex
- PyPI je místo, odkud `pip` stahuje všechny knihovny.
- Můžete zaregistrovat svůj vlastní package a zpřístupnit jej ke stažení vývojářům z celého světa.

Registrace projektu v PyPI

Chcete-li svůj balíček zaregistrovat v PyPI, musíte jej nejprve správně zabalit. **Předpoklady:**

1. **Python:** Nainstalován ve vašem systému.
2. **PyPI Účet:** Zaregistrován na pypi.org.
3. **Požadované Nástroje:** Nainstalujte nástroje pro balení:

```
pip install setuptools wheel build twine
```

Registrace projektu v PyPI

Krok 1: Struktura projektu a zdrojový kód

Vytvořte čistou adresářovou strukturu pro váš balíček.

```
my_awesome_package/
-- my_awesome_package/     <-- The actual source code package di
   -- __init__.py          <-- Needed for package
   -- simple_module.py     <-- This is your main function.
-- tests/
-- README.md               <-- Description for PyPI and GitHub
-- LICENSE
-- pyproject.toml          <-- New standard configuration file
```

Příklad v PyCharm.

Registrace projektu v PyPI

Krok 2: Vytvořte distribuci zdrojů a Wheel

Přejděte do **kořenového adresáře** vašeho projektu v terminálu a spusťte příkaz pro sestavení.

```
# This command creates two directories: 'build' and 'dist'
python -m build
```

Tento příkaz generuje distribuční soubory v nově vytvořeném adresáři `dist/`:

1. **Distribuce zdrojů (`.tar.gz`):** Používá se primárně pro starší instalace a nástroje pro balení.
2. **Wheel (`.whl`):** Předem sestavený distribuční formát pro rychlejší a spolehlivější instalaci.

Registrace projektu v PyPI

Krok 3: Publikování na Oficiální PyPI

Twine je nástroj pro publikování balíčků Pythonu na PyPI. Pokud má balíček správnou strukturu se všemi požadovanými soubory, lze jej zaregistrovat v PyPI.

```
1 twine upload dist/*
```

Twine vás vyzve k zadání oficiálního uživatelského jména a hesla (nebo API tokenu) k PyPI. Váš balíček je nyní veřejně dostupný a lze jej nainstalovat pomocí:

```
1 pip install my-awesome-package-your-username
```

Funkce a metody

Parametry funkce

Parametry funkce mohou být:

- vyžadované (povinné)
- volitelné (pojmenované parametry).

Argumenty pro povinné parametry se obvykle předávají bez uvedení jejich názvů.
Argumenty pro volitelné parametry se obvykle předávají s uvedením jejich názvů při volání funkce.

Parametry funkce – příklad

```python
# Function for printing the name and surname
def print_full_name(name, surname):
    print (f"{name} {surname}")

# Calling a function without specifying thr parameter names
print_full_name("Jon", "Snow")

# Function call with names of all parameters
print_full_name(name="Jon", surname="Snow")

# Calling the function with the names of the last parameter
print_full_name("Jon", surname="Snow")
```

Typy ve funkcích

Python vám dává možnost specifikovat typy argumentů a návratové typy. Syntaxe je podobná té, kterou jsme se naučili při vytváření proměnných:

```
1 def print_hello(text: str) -> None:
2     print(f"Hello {text}")
3
4
5 print_hello("world")
```

Parametry funkce – výchozí parametry

Výchozí argumenty jsou hodnoty, které jsou poskytnuty při definování funkcí. Tyto parametry se stávají volitelnými při volání funkcí. Pokud při volání funkcí poskytneme hodnotu pro výchozí argumenty, přepíše se výchozí hodnota.

```
1 # The definition of the function greet_by_name (name) with
      the default value of the name
2 def greet_by_name(name="World!"):
3     print(f"Hello, {name}")
4
5 # Calling the function greet_by_name (name) without an
      argument
6 greet_by_name() # Prints "Hello, World!"
7 # Calling the function greet_by_name (name) with "John" as
      the name argument
8 greet_by_name("John") # Prints 'Hello, John'
9 greet_by_name(name="John") # Prints 'Hello, John'
```

Funkce – návratové hodnoty

- Funkce Pythonu mohou vracet vypočítané hodnoty pomocí klíčového slova `return`.
- Pokud ve funkci není použito `return`, funkce vrátí hodnotu `None`.
- Funkce vždy něco vrátí!

```
1 def calculate_square(a):
2     return a * a
3
4 square = calculate_square(5)
5 print(square) # Prints 25
```

Funkce – návratové typy

Můžeme specifikovat typ návratových hodnot. K tomu můžeme použít znak -> a dvojtečku. Příklad může vypadat takto:

```
1 def get_hello(text: str) -> str:
2     return f"Hello {text}"
3
4
5 print(get_hello("world"))
```

Funkce s libovolným počtem argumentů

```
1  # Add two numbers
2  def add(a, b):
3      return a + b
4
5  # Add three numbers
6  def add(a, b, c):
7      return a + b + c
8
9  # Add four numbers
10 def add(a, b, c, d):
11     return a + b + c + d
```

Co když chce uživatel sečíst 10 čísel?

Funkce s libovolným počtem argumentů – *args

- Namísto vytváření funkcí s velkým počtem pozičních argumentů můžete přidat parametr *args.
- Argumenty předané uživatelem se vloží na tuple (n-tice) *args a jsou dostupné uvnitř funkce.

```
# Add any number of numbers
def add(*args):
    result = 0
    for arg in args:
        result += arg
    return result

print(add(1,2,3,4,5))  # Prints 15

# Prints the name and what the user gives
def print_name_and_something(name, *strings):
    print(f"First name: {name}")
    for string in strings:
        print(string)
```

Funkce s libovolným počtem argumentů – **kwargs

- Namísto vytváření funkcí s obrovským počtem pojmenovaných argumentů můžete přidat parametr **kwargs.
- Pojmenované argumenty zadané uživatelem se vloží do slovníku s názvem **kwargs a jsou následně dostupné uvnitř funkce.

```
1 # Add any number of ingredients
2 def add_ingredients(**kwargs):
3     result = 0
4     for key in kwargs:
5         result += kwargs[key]
6     return result
7
8 print(add_ingredients(eggs=3, spam=5, cheese=2))  # Will
    print 10
```

Funkce s libovolným počtem argumentů – *args a **kwargs

Libovolný počet neklíčových (*args) a klíčových (**kwargs) argumentů lze kombinovat do jedné funkce.

```
# Add any number of ingredients
def add_ingredients(*args, **kwargs):
    result = 0
    for arg in args:
        result += arg
    for key in kwargs:
        result += kwargs[key]
    return result

print(add_ingredients(1, 2, 3, eggs=3, spam=5, cheese=2)) # Will print 16
```

Vnořené funkce a Scope

- Vnořená funkce (nebo vnitřní funkce) je jednoduše funkce definovaná uvnitř jiné funkce (vnější funkce).
- Účel: Skrýt vnitřní funkci před vnějším rozsahem, aby byla dostupná pouze vnější funkci. To pomáhá při organizaci a zapouzdření kódu.
- Vnořené funkce striktně dodržují pravidlo **Local**, **Enclosing**, **Global**, **Built-in (LEGB)**.
  - Vnořená funkce má přístup k proměnným ve svém vlastním **Lokálním** rozsahu.
  - Má také přístup k proměnným v rozsahu **Vnější** (uzavírající) funkce.

Uzávěry (Closures)

- **Uzávěr** je vnořená funkce, která si pamatuje a přistupuje k proměnným ze svého vnějšího (uzavírajícího) rozsahu, i když byla vnější funkce dokončena.
- Mechanismus: Když vnější funkce vrátí vnořenou funkci, vnořená funkce nese odkaz na prostředí vnější funkce (její data).
- Praktické Použití: Uzávěry se často používají k vytváření **továrních funkcí** (funkcí, které generují a vracejí jiné přizpůsobené funkce) a k implementaci **dekorátorů**.

Příklad uzávěru (továrna funkcí)

```python
def make_multiplier(factor):
    # 'factor' is a variable in the enclosing scope

    def multiplier(number):
        # 'multiplier' is the nested function (the closure)
        # It references 'factor' from the outer scope
        return number * factor

    return multiplier  # Returns the nested function itself

# Create customized functions using the factory:
double = make_multiplier(2)
triple = make_multiplier(3)

print(double(10))  # Output: 20 (it "remembers" factor=2)
print(triple(10))  # Output: 30 (it "remembers" factor=3)
```

Funkce jako objekt

Python považuje funkce za „objekt“, což znamená, že je lze používat jako jakýkoli jiný datový typ (celá čísla, řetězce atd.).

- **Přiřazení k proměnné:** Funkci lze vložit do proměnné.
- **Předání jako argument:** Funkci lze předat jako argument jiné funkci (viz dále).
- **Návratová hodnota:** Funkci lze vrátit jako hodnotu (jak bylo ukázáno u uzávěrů).

Funkcionální prvky (funkce vyššího řádu)

Funkce vyššího řádu (HOF) je funkce, která buď přijímá jednu nebo více funkcí jako argumenty, nebo vrací funkci jako svůj výsledek.

- **Předávání funkcí jako argumentů:** HOF, jako jsou `map()`, `filter()` a `sorted()`, přijímají funkci jako argument pro aplikaci logiky na iterovatelný objekt.

Příklady funkcionálních prvků – Lambda funkce

Malé anonymní (bezejmenné) funkce definované v jediném výrazu pomocí klíčového slova `lambda`. Vynikající pro jednoduché, jednorázové argumenty HOF.

```
1 list(filter(lambda x: x > 5, [1, 6, 3, 8]))
```

Příklady funkcionálních prvků – map()

Aplikuje funkci na každou položku iterovatelného objektu a vrací objekt `map` (což je iterátor výsledků).

```
1 list(map(str.upper, ['a', 'b', 'c']))  # ['A', 'B', 'C']
```

Příklady funkcionálních prvků – `filter()`

Konstruuje iterátor z prvků iterovatelného objektu, pro které funkce vrátí true.

```
1 list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4]))  # [2, 4]
```

Příklady funkcionálních prvků – List/Dict Comprehensions

Ačkoli to nejsou HOF, jsou to idiomatické způsoby Pythonu pro provádění mapovacích a filtračních operací, často upřednostňované před `map()` a `filter()` s `lambda`.

```
1 [x * 2 for x in data if x > 0]
```

Globální, lokální a vázané proměnné (Scope)

Tato sekce podrobně popisuje, jak Python určuje, na kterou definici proměnné odkazuje část kódu při jejím provádění, primárně se řídí pravidlem **LEGB**. **Rozsah platnosti (Scope)** je oblast kódu, kde je proměnná přístupná. Python definuje čtyři primární rozsahy, kontrolované v určitém pořadí:

1. **L**okální (Local)
2. **E**nclosing (Vnější / Vázaný)
3. **G**lobální (Global)
4. **B**uilt-in (Vestavěný)

Lokální Rozsah (L)

- Nejvnitřnější rozsah. To zahrnuje proměnné definované uvnitř aktuální funkce nebo metody.
- Když je proměnné přiřazena hodnota uvnitř funkce (např. `x = 10`), je automaticky považována za lokální pro danou funkci, pokud není explicitně deklarována jinak.
- Lokální proměnné existují pouze po dobu provádění funkce.

```
1 def my_function():
2     # 'x' is local to my_function
3     x = 10
4     print(x)
5
6 # Attempting to access 'x' here would result in a NameError
```

Vnější / Vázaný Rozsah (E)

- Rozsah funkce, která **obsahuje** jinou funkci (vnořenou funkci). Proměnné se zde často nazývají **vázané** nebo **nelokální**.
- Vnořená funkce může **číst** proměnné z rozsahu své vnější funkce. To je klíčové pro **uzávěry** (jak bylo diskutováno v předchozí sekci).
- Klíčové slovo `nonlocal`: Chcete-li upravit proměnnou ve vnějším rozsahu (namísto vytvoření nové lokální proměnné), musíte explicitně použít klíčové slovo `nonlocal`.

Vnější / Vázaný Rozsah (E) – příklad

```
1  def outer_function():
2      message = "Hello" # Variable in the Enclosing scope
3
4      def inner_function():
5          # Read-only access: reads the 'message' from
               outer_function
6          print(message)
7
8      def changer_function():
9          # Modification: without 'nonlocal', 'count' would be
               a new local variable
10         nonlocal count
11         count += 1
12         print(f"Inner count: {count}")
13
14     count = 0
15     inner_function()
16     changer_function()
17     print(f"Outer count: {count}")
```

Globální Rozsah (G)

- Rozsah aktuálního modulu (`.py` soubor). Proměnné definované na nejvyšší úrovni skriptu nebo modulu jsou **globální**.
- Jakákoli funkce může **číst** globální proměnnou bez explicitní deklarace.
- Klíčové slovo `global` (zápis): Chcete-li **upravit** nebo přeřadit proměnnou definovanou v globálním rozsahu zevnitř funkce, musíte použít klíčové slovo `global`. Bez něj by Python vytvořil novou lokální proměnnou se stejným názvem, přičemž globální proměnná by zůstala nezměněna.

Globální Rozsah (G) – příklad

```
1  global_var = 100 # Variable in the Global scope
2
3  def read_global():
4      # Reading is fine
5      print(f"Reading: {global_var}")
6
7  def modify_global():
8      # MUST declare 'global_var' as global to change the top-
           level variable
9      global global_var
10     global_var += 10 # This changes the top-level variable
11
12 read_global()      # Output: Reading: 100
13 modify_global()
14 print(f"After modification: {global_var}") # Output: After
       modification: 110
```

Vestavěný Rozsah (B)

- Nejširší rozsah.
- To zahrnuje předdefinované funkce a konstanty, které jsou vždy k dispozici (např. `print()`, `len()`, `True`, `None`).
- Tyto názvy jsou vždy přístupné, pokud nejsou zastíněny (přepsány) definicí proměnné v jednom ze tří dalších rozsahů.

Souhrn Správy Rozsahu

Klíčové slovo / Kontext	Rozsah Přístupu	Účel	Vyžadováno Pro
Žádná deklarace	Lokální	Výchozí rozsah pro přiřazení.	*Čtení* globálních či vnějších proměnných.
`global`	Globální	Pro přístup a **změnu** proměnné na úrovni modulu.	*Zápis* (přiřazení) do globální proměnné.
`nonlocal`	Vnější	Pro přístup a **změnu** proměnné v bezprostředně uzavírající funkci.	*Zápis* (přiřazení) do vnější proměnné.

Metody vs. funkce: zásadní rozdíl

- **Funkce:** Samostatná jednotka kódu, definovaná mimo jakoukoli třídu. Operuje s daty, která jsou jí předána.
- **Metoda:** Funkce, která patří ke třídě a operuje s **daty konkrétního objektu (instance)**.
- **Vazba:** Metoda je vázána na instanci objektu, přes kterou je volána.

Parametr `self` (metody instance)

- První parametr standardní metody instance musí být `self` (ačkoli název `self` je pouze konvence, je v Pythonu striktně dodržován).
- `self` je odkaz na **instanci** třídy, na které byla metoda volána. Umožňuje metodě přístup a modifikaci **atributů** (dat) instance.
- Když voláte `obj.method(arg1)`, Python automaticky předá `obj` jako první argument (`self`) metodě. Vy předáváte pouze `arg1`.

Typy metod

Python podporuje tři typy metod, kategorizované podle toho, na čem operují:

1. Metody instance (výchozí):
   - Operují s **daty instance** (`self.attribute`).
   - Jako první argument přijímají `self`. (Toto je nejběžnější typ.)
2. Metody třídy:
   - Operují s **daty třídy** (např. konstanty nebo čítače na úrovni třídy).
   - Dekorovány pomocí `@classmethod`.
   - Jako první argument přijímají samotnou třídu, konvenčně zvanou `cls`.
   - Používá se pro alternativní konstruktory (způsoby, jak vytvořit instanci).
3. Statické metody:
   - Neoperují s instancí ani třídou.
   - Dekorovány pomocí `@staticmethod`.
   - Jsou to jen běžné funkce **logicky seskupené** v rámci třídy (nepřijímají žádný speciální první argument jako `self` nebo `cls`).
   - Používají se, když se funkce vztahuje ke třídě, ale nepotřebuje přístup k jejímu stavu.

Typy metod

- Pokud vaše logika potřebuje pracovat s jedinečnými daty objektu, použijte **metodu instance** (s `self`).
- Pokud vaše logika potřebuje manipulovat se samotnou třídou (např. vytvořit instanci z jiného datového formátu), použijte **metodu třídy** (s `cls`).
- Pokud vaše logika souvisí se třídou, ale je nezávislá na datech jakéhokoli konkrétního objektu, zvažte **statickou metodu**.

Pokročilé typování (type hinting)

- **Definice:** Způsob, jak deklarovat očekávané typy pro argumenty funkce, návratové hodnoty a proměnné.
- **Status:** Typové anotace jsou **volitelné** a nemají žádný vliv na chování kódu za běhu programu (Python je stále dynamicky typovaný jazyk).
- **Účel:**
  1. **Zlepšení čitelnosti:** Dokumentuje rozhraní funkcí (co funkce očekává a co vrací).
  2. **Nástroje IDE:** Umožňuje IDE (PyCharm, VS Code) poskytovat přesnější automatické doplňování kódu a kontroly chyb.
  3. **Statická analýza:** Klíčové pro použití nástrojů jako Mypy.

Type Hinting – základní syntaxe

```python
from typing import List, Optional

# Basic types for arguments and return values
def calculate_vat(price: float, rate: float) -> float:
    return price * rate

# Optional parameters
def greet(name: str, enthusiasm: bool = False) -> str:
    if enthusiasm:
        return f"Hello, {name}!"
    return f"Hello, {name}"

# Composite types (Collections)
def process_names(names: List[str]) -> None:
    for name in names:
        print(name.upper())
```

Type Hinting – základní syntaxe

```
1 from typing import List, Optional
2
3 # Type 'Optional' (Can be the specified type OR None)
4 def find_user(user_id: int) -> Optional[dict]:
5     if user_id > 1000:
6         return None
7     return {"id": user_id}
```

Statická analýza (mypy)

- **Instalace:** `pip install mypy`
- **Definice:** Mypy je statický analyzátor, který provádí kontrolu typů na základě type hinting anotací **bez spuštění kódu**.
- **Výhoda:** Zjistí chyby související s typy (např. pokus o předání řetězce funkci, která očekává celé číslo) dříve, než se kód spustí.
- **Použití:** Nástroj se spouští v terminálu nebo v CI/CD pipeline.
- **Spuštění:** `mypy my_module.py`

```python
# Example: Mypy detects an error
def add_numbers(a: int, b: int) -> int:
    return a + b

result = add_numbers(10, "20") # Mypy: Error! Expected int, got str.

# Running in terminal:
# mypy my_module.py
```

Statická analýza kódu (Linters)

Nástroje pro statickou analýzu („lintery“) kontrolují zdrojový kód bez jeho spuštění.

- **Účel:** Odhalují syntaktické chyby, nepoužité proměnné a vynucují stylistický standard **PEP 8**.
- `pylint`: Komplexní nástroj, kontroluje i návrhové vzory a složitost kódu.
- `flake8`: Populární, lehčí nástroj zaměřený na styl a syntaxi.
- `ruff`: Moderní, extrémně rychlá alternativa (napsaná v Rustu), která nahrazuje oba výše zmíněné.

```
1 # Example of code with issues (bad_style.py)
2 def ADD(x,y):         # Bad naming (caps), missing whitespace
3     res = x + y       # Unused variable 'res'
4     return x+y
5
6 # Running pylint:
7 tmp.py:4:0: C0304: Final newline missing (missing-final-
      newline)
```

Objektový model

Čtyři pilíře OOP (opakování)

Objektově orientovaný návrh je postaven na čtyřech základních principech:

1. **Zapouzdření (Encapsulation):** Sdružování **dat** (*atributů*) a **metod** (*funkcí*), které s těmito daty pracují, do jedné jednotky (třídy). Tím se zabraňuje vnějšímu, neautorizovanému přístupu a modifikaci vnitřního stavu.
2. **Abstrakce (Abstraction):** Skrývání složitých detailů implementace a zobrazování pouze nezbytných rysů objektu. Uživatelé interagují s jednoduchým rozhraním.
3. **Dědičnost (Inheritance):** Mechanismus, kterým jedna třída získává vlastnosti a metody jiné třídy (rodičovské nebo nadtřídy). To podporuje znovupoužitelnost kódu.
4. **Polymorfie (Polymorphism):** Schopnost různých objektů reagovat na stejné volání metody způsobem, který je specifický pro jejich vlastní typ.

Zvláštnosti Pythonu (The Pythonic Way)

Zatímco Python podporuje čtyři pilíře, jeho dynamická povaha podporuje několik charakteristických návrhových postupů:

1. Duck Typing (Kachní typování)
2. Kompozice nad dědičností (*Composition Over Inheritance* – COI)
3. Explicitní je lepší než implicitní

Zvláštnosti Pythonu – Duck Typing

- Python se primárně nestará o **typ** objektu (jaká je to třída), ale spíše o jeho **rozhraní** (jaké má metody a atributy).
- Motto: *'Pokud to chodí jako kachna a kváká jako kachna, pak je to kachna.'*
- Praktický dopad: Zaměřte se na poskytování nezbytných metod. Například jakýkoli objekt, který poskytuje metodu `__len__`, lze použít s vestavěnou funkcí `len()`, bez ohledu na hierarchii tříd.
- To vede k **vysoce flexibilním rozhraním**.

Zvláštnosti Pythonu – Kompozice nad dědičností (COI)

- Při navrhování systému upřednostňujte skládání objektů (pomocí atributů, které jsou instancemi jiných tříd – **kompozice**) před vytvářením hlubokých, rigidních hierarchií tříd (**dědičnost**).
- Dědičnost vytváří těsnou vazbu a může vést ke křehkým bázovým třídám. Kompozice umožňuje větší flexibilitu a snadnější testování.
- Tento princip je silně podporován **Mixins** (probráno později), což je forma kompozice prostřednictvím delegování.

Zvláštnosti Pythonu – Explicitní je lepší než implicitní

- Návrhová rozhodnutí by měla být jasná a přímá.
- Dopad na zapouzdření: Na rozdíl od jazyků, které vynucují striktní soukromé proměnné, Python používá konvenci: atributy s předponou jednoho podtržítka (`_attribute`) naznačují, že jsou **zamýšleny jako soukromé** (chráněné).
- Python však spoléhá na disciplínu programátora, nikoli na vynucenou syntaxi, aby tuto konvenci respektoval. Toto se někdy nazývá „konsensuální programování“.

Shrnutí pro návrh

Efektivní OOD v Pythonu znamená navrhování tříd a modulů, které:

- Definují **jasná rozhraní** (metody a dunder metody).
- Jsou flexibilní díky **Duck Typingu**.
- Upřednostňují **Mixins** a **Kompozici** pro kombinování funkcionality.
- Jsou **snadno čitelné** a udržovatelné, dodržují standardní konvence (jako použití _ pro chráněné atributy).

Identita objektu, rovnost a reference

Je důležité pochopit rozdíl mezi hodnotou objektu a jeho umístěním v paměti:

- **Identita (`is`):** Kontroluje, zda dvě proměnné odkazují na **přesně stejný objekt** v paměti.
  (Kontrolováno pomocí vestavěné funkce `id()`).
- **Rovnost (`==`):** Kontroluje, zda mají dvě proměnné **stejnou hodnotu**.
  (Používá speciální metodu `__eq__`).
- **Aliasing:** Když více proměnných odkazuje na stejný měnitelný objekt. Změna objektu přes jeden alias jej změní pro všechny ostatní.

Identita objektu, rovnost a reference – Příklad

```
# Example: Immutability vs. Mutability
a = [1, 2]
b = a           # Mutable: b is an alias for a
a.append(3)     # Changes the object in place
print(b)        # Output: [1, 2, 3]

c = (1, 2)
d = c
c += (3,)       # Immutable: This creates a NEW tuple 'c'
print(d)        # Output: (1, 2)  (d still references the
    original object)
```

Hashovatelnost

- Hashovatelný objekt je takový, který má hash hodnotu, která se **nikdy nemění** během jeho životnosti (tj. jeho metoda `__hash__` je definována a vrací konstantní hodnotu).
- Pouze **neměnné** objekty jsou hashovatelné. Měnitelné objekty *nejsou* hashovatelné.
- Hashovatelné objekty jsou povinné pro použití jako **klíče ve slovnících** (`dict`) a jako **členy množin** (`set`), protože tyto datové struktury spoléhají na konstantní hash hodnotu pro efektivní vyhledávání.

Speciální metody (Dunder metody)

Speciální metody jsou srdcem objektového modelu Pythonu a umožňují vašim vlastním objektům bezproblémově spolupracovat s klíčovými funkcemi jazyka, operátory a vestavěnými funkcemi. Definují *protokol* nebo *rozhraní* objektu.

1. Základní identita a reprezentace
2. Přetěžování operátorů (Emulace číselných typů)
3. Protokoly kontejnerů (Emulace kolekcí)

I. Základní identita a reprezentace

Tyto metody řídí, jak je objekt vytvářen a jak je zobrazován.

- `__init__(self, ...)`
  - **Konstruktor:** Inicializuje nově vytvořený objekt. Nastavuje počáteční stav instance (atributy). `obj = MyClass(arg1)`
- `__new__(cls, ...)`
  - **Vytvoření:** Voláno pro vytvoření nové instance před `__init__`. Zřídka přepisováno, typicky pouze pro pokročilé použití jako Singletony nebo neměnné třídy.
- `__del__(self)`
  - **Destruktor:** Volán, když má být instance zničena (garbage collected). Zřídka používáno, protože Python spravuje paměť automaticky.

I. Základní identita a reprezentace

Tyto metody řídí, jak je objekt vytvářen a jak je zobrazován.

- `__str__(self)`
  - **Neformální řetězcová reprezentace:** Vrací „uživatelsky přívětivý“ řetězec určený pro zobrazení koncovému uživateli (např. v `print()`).
    `print(obj)`
- `__repr__(self)`
  - **Oficiální řetězcová reprezentace:** Vrací řetězec, který by měl být jednoznačný a ideálně reprodukovatelný, často používaný pro ladění a logování. Pokud není definováno `__str__`, Python použije `__repr__`.
    `repr(obj)`

II. Přetěžování operátorů (Emulace číselných typů)

Tyto metody umožňují vlastním objektům reagovat na standardní matematické, porovnávací a logické operátory.

- `__add__(self, other)`
  - Operátor: `+`
  - Implementuje sčítání. `a + b`
- `__sub__(self, other)`
  - Operátor: `-`
  - Implementuje odčítání. `a - b`
- `__mul__(self, other)`
  - Operátor: `*`
  - Implementuje násobení. `a * b`

II. Přetěžování operátorů (Emulace číselných typů)

- `__eq__(self, other)`
  - Operátor: `==`
  - Definuje kontrolu rovnosti. Nezbytné pro správné porovnávání. `a == b`
- `__lt__(self, other)`
  - Operátor: `<`
  - Definuje kontrolu menší než. `a < b`
- `__bool__(self)`
  - Operátor: `bool()`
  - Definuje pravdivostní hodnotu objektu (např. zda je objekt považován za prázdný nebo False). Pokud není definováno, Python kontroluje `__len__`. `if obj:`

III. Protokoly kontejnerů (Emulace kolekcí)

Tyto metody umožňují objektům chovat se jako sekvenční typy (`list`, `tuple`) nebo mapovací typy (`dict`).

- `__len__(self)`
  - `len()`
  - Vrací délku nebo velikost kontejneru. `len(obj)`
- `__getitem__(self, key)`
  - `[]` (indexování/řezání)
  - Získává položku nebo řez pomocí závorkové notace. `obj[key]`
- `__setitem__(self, key, value)`
  - `[]` (přiřazení)
  - Nastavuje hodnotu na daném indexu nebo klíči. `obj[key] = value`

III. Protokoly kontejnerů (Emulace kolekcí)

- `__contains__(self, item)`
  - `in`
  - Definuje chování pro testování členství. `item in obj`
- `__iter__(self)`
  - Iterace
  - Vrací objekt iterátoru, což umožňuje použití objektu v cyklu `for`. `for item in obj:`
- `__next__(self)`
  - Iterace
  - (Definováno v objektu iterátoru vráceném `__iter__`) Vrací další položku z kontejneru. (Používáno implicitně cyklem `for`)

Návrh řízený protokoly

Síla Dunder metod spočívá v **Duck Typingu** a **návrhu řízeném protokoly**. Implementací specifické sady dunder metod (protokolu) získá vaše třída všechny související funkce Pythonu, takže se chová jako nativní typ, aniž by musela dědit z nějakého konkrétního objektu.
Například jakákoli třída, která definuje `__len__` a `__getitem__`, se automaticky chová jako sekvence.

Deskriptory (Protokol přístupu k atributům)

Deskriptory jsou mechanismem, který stojí za mnoha nejdůležitějšími objektově orientovanými funkcemi Pythonu, včetně metod, `@classmethod`, `@staticmethod`, `@property` a dokonce i implementace `__slots__`.

I. Definice deskriptoru a protokol

- Deskriptor je jakýkoli objekt, který implementuje alespoň jednu z metod protokolu deskriptoru (`__get__`, `__set__` nebo `__delete__`).
- Instance deskriptoru musí být definována jako **třídní proměnná** ve třídě vlastníka.
- Když je přistupováno k atributu, Python zachytí požadavek a deleguje akci na odpovídající metodu objektu deskriptoru.
- `__get__(self, instance, owner)` – **Getter:** Získává hodnotu atributu. Atribut je čten (např. `obj.attr`).
- `__set__(self, instance, value)` – **Setter:** Nastavuje nebo přiřazuje hodnotu atributu. Atribut je přiřazován (např. `obj.attr = value`).
- `__delete__(self, instance)` – **Deleter:** Maže atribut. Atribut je mazán (např. `del obj.attr`).

Klíčové parametry:

- `instance`: Konkrétní instance třídy vlastníka, ke které se přistupuje (nebo `None`, pokud se přistupuje přes třídu).

II. Typy deskriptorů

Deskriptory se dělí podle toho, které metody implementují, což ovlivňuje pořadí vyhledávání atributů:

1. **Datové deskriptory:** Implementují jak `__get__`, tak `__set__` (nebo `__delete__`).
   - Mají vždy přednost před instančními atributy (proměnnými uloženými v `__dict__` instance). To je nezbytné pro implementaci funkcí jako `@property`.
2. **Nedatové deskriptory:** Implementují pouze `__get__`.
   - Mohou být **zastíněny** (přepsány) instančními atributy. Pokud má instance atribut se stejným názvem, vrátí se instanční atribut místo hodnoty deskriptoru. (Metody jsou nedatové deskriptory).

III. Praktické využití: Dekorátor `@property`

Nejběžnější způsob použití deskriptorů je prostřednictvím vestavěného dekorátoru `@property`. Umožňuje transformovat standardní instanční atribut na vlastnost s řízeným přístupem, která se používá pro:

- **Validaci:** Zajištění, že přiřazené hodnoty splňují kritéria.
- **Vypočítané atributy:** Výpočet hodnoty za běhu (např. věk z data narození).
- **Atributy pouze pro čtení:** Definování pouze getteru (`@property`) bez setteru (`@attr.setter`).

Příklad @property (Validace)

```
1  class Temperature:
2      def __init__(self, celsius):
3          self.celsius = celsius # Calls the setter
4
5      @property # GETTER: Defines the method for reading the
           attribute
6      def celsius(self):
7          return self._celsius
8
9      @celsius.setter # SETTER: Defines the method for writing
           the attribute
10     def celsius(self, value):
11         # Validation logic (the power of the descriptor
               protocol)
12         if value < -273.15: # Absolute zero
13             raise ValueError("Temperature below absolute zero
                   is impossible.")
14         self._celsius = value # Stores the value in a "
               private" attribute
```

Příklad @property (Validace)

```
# Usage
temp = Temperature(25)
print(temp.celsius) # Calls the @property getter

# This call triggers the @celsius.setter method
temp.celsius = 30

# This would raise a ValueError due to the descriptor logic
# temp.celsius = -300
```

IV. Shrnutí

Deskriptory jsou protokolem, který definuje, **co se stane, když přistoupíte k atributu**. Pochopení deskriptorů pomáhá vysvětlit, proč volání metody na instanci automaticky zahrnuje argument `self`, a jak nástroje jako `@property` mohou spouštět vlastní kód během jednoduchého přiřazení (`=`) nebo přístupu (`.`).

Dědičnost

- Proces, kdy **Podtřída** (nebo odvozená třída/dceřiná třída) získává vlastnosti a metody **Nadtřídy** (nebo bázové třídy/rodičovské třídy). Modeluje vztah „**je**“ (např. Pes *je* Savec).
- Podporuje **znovupoužitelnost kódu** a definuje hierarchii typů.

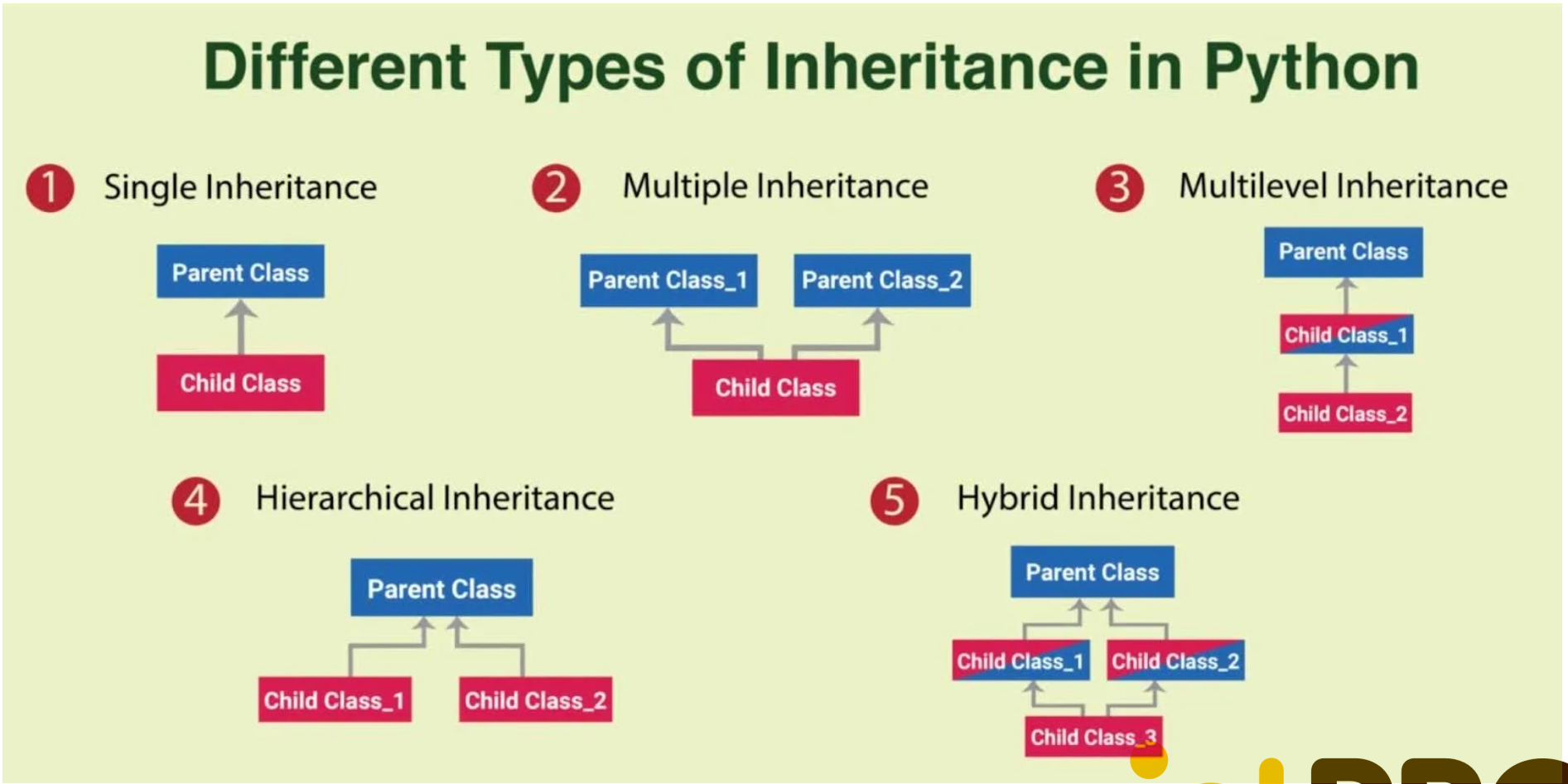
```
1 class Parent:  # Superclass
2     # ...
3
4 class Child(Parent):  # Subclass inherits Parent
5     # ...
```

Dědičnost

- **Přepisování metod (Overriding):** Podtřída může předefinovat metodu zděděnou z nadtřídy a poskytnout specializovanou implementaci.
- Funkce `super()`:
  - Způsob, jak přistupovat k metodám a atributům rodiče (nadtřídy) zevnitř podtřídy.
  - Nejčastěji se používá v metodě `__init__` podtřídy, aby se zajistilo provedení inicializační logiky rodiče a zabránilo se ztrátě atributů z rodičovské třídy.

```
1 class Animal:
2     def __init__(self, species):
3         self.species = species
4
5 class Dog(Animal):
6     def __init__(self, species, breed):
7         # Calls Animal.__init__(self, species)
8         super().__init__(species)
9         self.breed = breed
```

Dědičnost – typy dědičnosti



Polymorfie

- Schopnost různých objektů reagovat na stejné volání metody způsobem, který je specifický pro jejich typ nebo třídu.
- Umožňuje psát obecný kód, který může pracovat s objekty různých tříd, pokud tyto objekty sdílejí společné rozhraní.

Polymorfie v Pythonu (Duck Typing)

Polymorfie v Pythonu je typicky dosahována nikoli prostřednictvím rigidních rozhraní (jako v Javě nebo C++), ale prostřednictvím **Duck Typingu**:

- Pokud dvě různé, nesouvisející třídy definují metodu se stejným názvem (např. `make_sound()`), jsou považovány za polymorfní vzhledem k této metodě.
- Příklad: Vestavěná funkce `len()` je polymorfní. Nezajímá ji, zda počítá `list`, `tuple` nebo `string`; prostě zavolá speciální metodu objektu `__len__`.

Polymorfie v Pythonu (Duck Typing) – příklad

```python
def describe_creature(creature):
    # This function works with any object that has a 'make_sound' method
    print(f"The creature says: {creature.make_sound()}")

class Cat:
    def make_sound(self):
        return "Meow"

class Dog:
    def make_sound(self):
        return "Woof"

describe_creature(Cat())  # Works
describe_creature(Dog())  # Works
```

Shrnutí

- **Dědičnost**
  - Znovupoužití kódu a hierarchie tříd.
  - **Struktura:** Definuje, čím objekt *je* (Is-a).
- **Polymorfie**
  - Přizpůsobivá, flexibilní rozhraní.
  - **Chování:** Definuje, co objekt *umí dělat* (Can-do).

Vícenásobná dědičnost (Multiple Inheritance – MI)

Třída dědící vlastnosti a metody ze **dvou nebo více** rodičovských (bázových) tříd.

```
1 class Child(ParentA, ParentB):
2     pass
```

Diamond Problem (Problém diamantu): Hlavní výzvou v MI je, jak řešit volání metod, když stejný název metody existuje ve více rodičovských třídách, zejména pokud tito rodiče sdílejí společného předka (vytvářejí tvar „diamantu“ v grafu hierarchie tříd).

Method Resolution Order (MRO)

- Python používá **Method Resolution Order (MRO)** k určení pořadí, ve kterém jsou prohledávány bázové třídy pro metodu nebo atribut.
- Python používá sofistikovaný algoritmus **C3 Linearization** k definování MRO, který zaručuje, že:
  - Třída je vždy prohledávána před svými rodiči.
  - Je respektováno pořadí bázových tříd v definici třídy (`class Child(ParentA, ParentB)`).
- MRO můžete explicitně zkontrolovat pro pochopení pořadí volání:

```
1 print(Child.mro())
2 # Or:
3 print(Child.__mro__)
```

- Při práci s MI `super()` **nevolá** jednoduše přímého rodiče; volá další třídu v řetězci MRO. To je zásadní pro kooperativní použití metod napříč hierarchií.

Mixins (Kompozice s vícenásobnou dědičností)

- *Mixin* je třída navržená k poskytování specifické, diskrétní sady metod (funkcionality) jiným třídám. Obecně není určena k samostatné instanciaci.
- Mixins využívají vícenásobnou dědičnost k dosažení **kompozice nad dědičností**. Umožňují třídě **'něco dělat'**, aniž by to znamenalo vztah **'je'**.

Mixins (Kompozice s vícenásobnou dědičností)

- **Návrhové principy pro Mixins:**
  1. **Jedna odpovědnost:** Mixin by měl poskytovat jedno specifické chování (např. logování, serializaci, validaci).
  2. **Bezstavovost:** Obecně by **neměly** definovat metodu `__init__` ani mít vlastní instanční atributy, protože jsou určeny k manipulaci se stavem třídy, do které jsou přimíchány.
  3. **Jmenná konvence:** Často mají příponu `Mixin` (např. `LoggingMixin`, `JSONSerializerMixin`) pro přehlednost.
- Mixins jsou typicky uvedeny **před** hlavní bázovou třídou v seznamu definice třídy, což zajišťuje, že jejich metody jsou v případě potřeby nalezeny v MRO jako první.

Příklad Mixinu (Přidání schopnosti logování)

```python
class LoggingMixin:
    """Provides a reusable method for logging events."""
    def log(self, message):
        import datetime
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        print(f"[{timestamp}] LOG: {message}")

# User inherits core identity from BaseUser and functionality from LoggingMixin
class User(LoggingMixin, BaseUser):
    def save(self):
        # The User class gains the 'log' method from the Mixin
        self.log(f"Saving user {self.username} to database.")

        # ... actual save logic
```

Shrnutí

- **Jednoduchá dědičnost**
  - Definuje hierarchii objektů (Is-a).
  - Striktní hierarchie.
  - `super()` pro volání rodiče.
- **Mixins**
  - Přidávají modulární funkcionalitu (Can-do).
  - Kompozice/Delegování.
  - Vícenásobná dědičnost + MRO.

Základní návrhové vzory

Návrhové vzory jsou formalizované osvědčené postupy, které může programátor použít k řešení běžných problémů při návrhu softwaru. Jsou to opakovaně použitelná řešení opakujících se návrhových problémů ve vývoji objektově orientovaného softwaru. Kategorie – Následující kategorizace je převzata z knihy **Gang Of Four** (Design Patterns: Elements of Reusable Object-Oriented Software):

- **kreacionální** vzory – jak **vytvářet** objekty
- **strukturální** vzory – jak popsat **strukturu** souvisejících objektů
- **behaviorální** vzory – jak naprogramovat **chování a odpovědnosti** souvisejících objektů.

Kreacionální vzory

- Singleton
- Builder
- Prototype
- Abstract Factory
- Factory Method

Kreacionální vzory – The Factory Pattern

Factory Pattern (Tovární vzor) je jedním z nejběžnějších a nejjednodušších vzorů, zvláště vhodný pro Python díky jeho dynamické povaze.

- Poskytuje rozhraní pro vytváření objektů v nadtřídě, ale umožňuje podtřídám (nebo jediné funkci) měnit typ objektů, které budou instanciocány.
- Odděluje logiku vytváření objektů od kódu klienta, který objekt používá. Klient požádá „továrnu“ o objekt na základě parametru a továrna rozhodne, kterou konkrétní třídu vrátí.
- Flexibilita: Snadné přidávání nových typů produktů bez úpravy kódu klienta.
- Zapouzdření: Složitost vytváření objektů je skryta uvnitř továrny.

Kreacionální vzory – The Factory Pattern

V Pythonu je Factory Pattern často implementován pomocí jednoduché tovární funkce namísto složitých hierarchií tříd.

```python
# Product classes
class Car:
    def drive(self):
        return "Vroom! Driving a Car."
class Bicycle:
    def drive(self):
        return "Cycling gracefully."
# The Factory Function
def vehicle_factory(type: str):
    """Factory function to create different vehicle objects.
       """
    if type.lower() == 'car':
        return Car()
    elif type.lower() == 'bike':
        return Bicycle()
    else:
        raise ValueError(f"Unknown vehicle type: {type}")
```

Strukturální vzory

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Strukturální vzory – The Decorator Pattern

Tento nívrhový vzor je v Pythonu velmi důležitý, protože Python poskytuje vestavěnou syntaxi `@decorator`.

- Slouží k **dynamickému připojení dalších odpovědností** k objektu. Dekorátory poskytují flexibilní alternativu k podtřídám pro rozšiřování funkcionality.
- Obalí původní objekt (nebo funkci) jiným objektem (dekorátorem), který přidává chování před a/nebo po volání originálu.
- Funkcionalitu lze přidat nebo odebrat za běhu. Dodržuje **Open/Closed Principle** (otevřeno pro rozšíření, uzavřeno pro modifikaci).

Strukturální vzory – The Decorator Pattern

```python
def log_execution_time(func):
    """Decorator to measure the execution time of a function.
    """
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs) # Call the original function
        end = time.time()
        print(f"[{func.__name__}] executed in: {end - start:.4f}s")
        return result
    return wrapper

@log_execution_time
def complex_calculation(n):
    # Imagine this is slow calculation logic
    sum(range(n))

# When complex_calculation is called, it is wrapped by the '
```

Behaviorální vzory

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Behaviorální vzory – The Strategy Pattern

- Definuje rodinu algoritmů, zapouzdří každý z nich do samostatné třídy a učiní jejich objekty zaměnitelnými.
- Zapouzdřuje zaměnitelné chování (strategie) a nechává klienta vybrat, kterou použít za běhu.
- Umožňuje vybírat chování dynamicky a nezávisle na klientovi, který je používá.

Behaviorální vzory – The Strategy Pattern

Definování Strategií (Algoritmů)

```python
class UsdFormatStrategy:
    """Strategy to format price in USD."""
    def format_price(self, amount: float) -> str:
        return f"${amount:,.2f}"

class EurFormatStrategy:
    """Strategy to format price in EUR."""
    def format_price(self, amount: float) -> str:
        # Note: European format often uses comma as decimal separator
        return f"EUR{amount:,.2f}".replace(",", "_").replace(".", ",").replace("_", ".")

class CzkFormatStrategy:
    """Strategy to format price in CZK."""
    def format_price(self, amount: float) -> str:
        return f"{amount:,.0f} CZK".replace(",", " ")  # Simple space thousands separator
```

Behaviorální vzory – The Strategy Pattern

Definování Kontextu (Třída, která používá Strategii)

```
1  class Product:
2      def __init__(self, name: str, price: float,
           format_strategy):
3          self.name = name
4          self.price = price
5          # The Context holds a reference to the Strategy
               object
6          self._formatter = format_strategy
7
8      # Setter method to change the strategy at runtime
9      def set_format_strategy(self, new_strategy):
10         self._formatter = new_strategy
11
12     # The method that uses the current strategy
13     def get_formatted_price(self) -> str:
14         # The Context delegates the task to the Strategy
15         return self._formatter.format_price(self.price)
```

Behaviorální vzory – The Strategy Pattern

Použití: Výběr a Změna Strategií

```
# Create the strategies
usd_formatter = UsdFormatStrategy()
eur_formatter = EurFormatStrategy()
czk_formatter = CzkFormatStrategy()

# 1. Initialize the product with the USD strategy
product = Product("High-End Monitor", 899.99, usd_formatter)

print(f"Product: {product.name}")
print(f"Price (Initial USD): {product.get_formatted_price()}")
# Output: Price (Initial USD): $899.99

# 2. Change the strategy at runtime (e.g., based on user's location)
product.set_format_strategy(eur_formatter)
print(f"Price (Changed to EUR): {product.get_formatted_price()}")
```

Generátory a iterátory

Sekvenční datové typy

Sekvenční datové typy jsou seřazené kolekce prvků. Všechny podporují běžné operace jako indexování, slicing (řezání), iteraci a testování členství (`in`). Hlavní rozdíl mezi nimi spočívá v měnitelnosti.
Běžné sekvenční operace:

- **Indexování:** Přístup k jednotlivým prvkům pomocí jejich pozice založené na nule (např. `my_list[0]`, `my_string[-1]`).
- **Slicing (Řezání):** Extrakce podsekvence pomocí syntaxe `[start:stop:step]` (např. `my_list[1:4]`).
- **Iterace:** Podpora cyklu `for item in sequence:` implementací iteračního protokolu.
- **Konkatenace (Spojování):** Kombinace dvou sekvencí stejného typu pomocí operátoru `+`.
- **Opakování:** Opakování sekvence pomocí operátoru `*` (např. `[1, 2] * 3` dává `[1, 2, 1, 2, 1, 2]`).
- **Délka:** Zjištění počtu prvků pomocí `len(sequence)`.

Měnitelné sekvence: Seznamy (`list`)

Seznamy jsou nejuniverzálnějším a nejčastěji používaným typem měnitelné sekvence.

- **Měnitelnost:** Jejich obsah **lze měnit** na místě po vytvoření (např. přidávání, odebírání nebo nahrazování prvků).
- **Charakteristiky:**
  - **Heterogenní:** Mohou obsahovat prvky různých datových typů (např. `[1, "hello", 3.14]`).
  - **Dynamická velikost:** Mohou snadno růst nebo se zmenšovat.
- **Klíčové operace (metody, které mění na místě):**
  - `append(x)`: Přidá `x` na konec.
  - `insert(i, x)`: Vloží `x` na specifický index `i`.
  - `remove(x)`: Odstraní první výskyt `x`.
  - `pop(i)`: Odstraní a vrátí prvek na indexu `i`.
  - `sort()`: Seřadí seznam na místě (modifikuje původní seznam).

Neměnné sekvence

Neměnné sekvence nelze po vytvoření modifikovat. Jakákoli operace, která vypadá, že je mění, ve skutečnosti vede k vytvoření nového objektu.

1. **N-tice (`tuple`)**
   - **Neměnné.** Díky tomu jsou pro iteraci a vyhledávání rychlejší než seznamy.
   - **Hashovatelné:** Protože jsou neměnné, lze n-tice použít jako **klíče ve slovnících** a jako **prvky v množinách**.
   - **Případy užití:** Často se používají pro definování konstantních skupin souvisejících dat (např. souřadnice: `(x, y)`) nebo pro funkce vracející více hodnot.
2. **Řetězce (`str`)**
   - **Neměnné.** Jakákoli operace jako `s.upper()` vrací nový řetězec.
   - Sekvence Unicode znaků.
3. **Rozsahy (`range`)**
   - Reprezentují neměnnou sekvenci čísel.
   - Jsou vysoce **paměťově efektivní**, protože ukládají pouze hodnoty `start`, `stop` a `step`, a generují sekvenci na vyžádání místo explicitního ukládání všech čísel.

Neměnné sekvence

Typ	Měnitelnost	Hashovatelné?	Typický případ užití
`list`	Měnitelné	Ne	Obecné kolekce, dynamické datové struktury.
`tuple`	Neměnné	Ano	Klíče slovníků, fixní záznamy, návratové hodnoty funkcí.
`str`	Neměnné	Ano	Textová data.
`range`	Neměnné	Ano	Iterování přes číselné sekvence (v cyklech for).

Generované sekvence

Generované sekvence jsou kolekce, které produkují své prvky jeden po druhém, na vyžádání. Tento přístup, známý jako **líné vyhodnocování (lazy evaluation)**, je klíčový pro efektivní zpracování velkých nebo nekonečných datových sad tím, že udržuje spotřebu paměti konstantní.

1. Iterační protokol
2. Generátory (Pythonic Iterátor)

Iterační protokol

Schopnost generovat sekvenci na vyžádání je založena na **Iteračním protokolu**, který definuje dvě základní role:

1. **Iterable (Iterovatelné):** Jakýkoli objekt, který lze předat vestavěné funkci `iter()`. Tato funkce musí zavolat metodu objektu `__iter__`, která vrací objekt **Iterátor**. (Příklady: seznamy, řetězce, n-tice, slovníky).
2. **Iterator (Iterátor):** Jakýkoli objekt, který definuje následující dvě speciální metody:
   - `__iter__(self)`: Vrací samotný objekt iterátoru.
   - `__next__(self)`: Vrací další položku ze sekvence. Pokud nejsou k dispozici žádné další položky, **vyvolá výjimku** `StopIteration`.

Generátory (Pythonic Iterátor)

Generátory jsou nejběžnějším a nejidiomatičtějším způsobem, jak v Pythonu vytvářet vlastní iterátory bez psaní formální struktury třídy.
Generátorové funkce:

- Funkce, která obsahuje klíčové slovo `yield` místo `return`.
- Když je generátorová funkce zavolána, **nevykoná** své tělo okamžitě;
- místo toho vrátí **objekt generátoru** (který automaticky implementuje Iterační protokol).
- Klíčové slovo `yield` **pozastaví** provádění funkce a odešle hodnotu zpět volajícímu. Když volající požádá o další položku (pomocí `next()`), provádění pokračuje přesně tam, kde skončilo, včetně všech lokálních proměnných.

Generátory – příklad

```
1 def simple_generator():
2     yield "Start"
3     yield 2
4     yield 3
5
6 # Execution only starts upon calling next()
7 gen = simple_generator()
8 print(next(gen)) # Output: Start
9 print(next(gen)) # Output: 2
```

Generátorové výrazy

- Stručná, jednořádková syntaxe pro vytváření generátorů, podobná list comprehensions, ale používající **kulaté závorky** `()`.
- **Syntaxe:** `(expression for item in iterable if condition)`
- Jsou líně vyhodnocované a vysoce paměťově efektivní, což je činí vhodnějšími než list comprehensions, když nepotřebujete celou sekvenci v paměti najednou (např. při předávání funkci jako `sum()` nebo `any()`).

```
# List Comprehension (creates a list of 1 million numbers in RAM)
# huge_list = [x * 2 for x in range(1000000)]

# Generator Expression (creates a generator object; minimal RAM usage)
huge_gen = (x * 2 for x in range(1000000))

# The sum() function consumes the generated values one by one
total = sum(huge_gen)
```

Praktické výhody generátorů

- **Paměťová efektivita**: Největší výhoda. Ideální pro velké soubory, síťové streamy nebo nekonečné sekvence, protože v paměti drží pouze aktuální stav a poslední vygenerovanou hodnotu.
- **Líné vyhodnocování**: Výpočet se provádí pouze tehdy, když je hodnota explicitně vyžádána, což může ušetřit čas CPU, pokud není potřeba celá sekvence.
- **Budování pipeline**: Generátory lze snadno řetězit dohromady a vytvářet tak zpracovatelské pipeliny, kde výstup jednoho generátoru je vstupem dalšího, což optimalizuje využití zdrojů.

Čtení textových souborů (I/O a efektivita)

Práce se soubory je klasickou a praktickou aplikací konceptů Iterátoru a Generátoru, zejména při práci s velkými datovými sadami. Hlavním cílem je zpracovávat soubory **řádek po řádku** bez načítání celého obsahu do paměti najednou.

Standardní souborový protokol

V Pythonu je souborový objekt (vrácený vestavěnou funkcí `open()`) sám o sobě **iterátor**.

- Když je souborový objekt použit v cyklu, automaticky generuje (`yields`) jeden řádek souboru najednou a po každém generování posune interní ukazatel souboru vpřed.
- Tento proces zajišťuje, že v aktivní paměti je držen pouze jeden řádek (řetězec), což z něj činí nejefektivnější způsob čtení masivních souborů (např. gigabajtové logy nebo CSV).

Kontextový manažer (`with open(...)`)

Příkaz `with open(...)` používá **protokol kontextového manažera** (speciální metody `__enter__` a `__exit__`) pro manipulaci se souborem.

- `__enter__`: Otevře soubor a vrátí souborový objekt.
- `__exit__`: **Zaručuje**, že metoda `close()` souboru je zavolána při opuštění bloku, i když uvnitř bloku dojde k chybám (výjimkám). To zabraňuje úniku zdrojů a poškození přístupu k souborům.

Standardní souborový protokol – příklad

```python
def process_file_efficiently(filepath):
    """Reads a file line-by-line using the file object as an iterator."""
    try:
        # 'with open(...)' ensures the file is closed automatically
        with open(filepath, 'r', encoding='utf-8') as file:
            for line in file:
                # 'line' contains the line content, including the trailing newline character
                processed_line = line.strip().upper()
                print(processed_line)
    except FileNotFoundError:
        print(f"Error: File not found at {filepath}")

# This operation is memory-safe even for huge files.
# process_file_efficiently("massive_data.log")
```

Neefektivní vs. efektivní čtení

- `file.read()`
  - Načte **celý obsah souboru** do jediného řetězce v paměti.
  - Neefektivní.
  - Může způsobit `MemoryError` u velkých souborů.
- `file.readlines()`
  - Načte celý obsah souboru do **seznamu řetězců** (jeden řetězec na řádek).
  - Neefektivní.
  - Načte celou strukturu seznamu do paměti.
- `for line in file:`
  - Čte a generuje jeden řádek najednou.
  - **Vysoce efektivní**.
  - Standardní, „Pythonic“ přístup pro paměťově bezpečnou iteraci.

Použití vlastního generátoru pro pokročilé filtrování

Pro komplexní zpracování dat (např. filtrování, čištění nebo parsování) je idiomatické zabalit iteraci souboru do **vlastní generátorové funkce**. To vytváří znovupoužitelnou komponentu pipeliny.

- **Výhoda**: Čtení souboru zůstává líné a filtrační logika je aplikována líně, řádek po řádku.

Použití vlastního generátoru pro pokročilé filtrování – příklad

```python
def clean_and_filter_data(filepath, keyword):
    """
    A generator that yields only lines containing a specific
    keyword after stripping whitespace.
    """
    with open(filepath, 'r') as f:
        for line in f:
            stripped_line = line.strip()

            # Filtering logic applied lazily
            if keyword in stripped_line:
                yield stripped_line # Yields the clean,
                    filtered result

# The file is processed only as needed when the generator is
    consumed
for record in clean_and_filter_data("raw_data.txt", "ERROR"):
    # 'record' is ready for immediate processing
    print(f"Found error record: {record}")
```

Ukládání a zpracování dat

Ukládání objektů (Serializace)

Serializace je proces převodu komplexních datových struktur (jako jsou objekty, seznamy a slovníky) do proudu bytů nebo řetězcového formátu, který lze uložit do souboru nebo odeslat přes síť. Opačný proces se nazývá **deserializace**.
Klíčové pojmy:

- **Serializace**: Převod stavu objektu do uložitelného formátu.
- **Deserializace (Unpickling/Parsingu)**: Rekonstrukce původního objektu ze serializovaného formátu.

Nástroje Pythonu pro serializaci

Python nabízí několik modulů pro serializaci, z nichž každý je vhodný pro jiný případ použití.

- `pickle`
  - Binární formát.
  - Ukládání **objektů specifických pro Python** (komplexní typy, vlastní třídy).
  - `pickle.dump(obj, file)` / `pickle.load(file)`
- `json`
  - Textový formát (JSON).
  - **Interoperabilita** s jinými jazyky a webovými aplikacemi.
  - `json.dump(obj, file)` / `json.load(file)`
- `csv`
  - Textový formát (CSV).
  - Jednoduchá strukturovaná **tabulková data** (řádky a sloupce).
  - Používá `csv.writer` a `csv.reader`.

Modul `pickle` (Python-specifická serializace)

`pickle` je nativní způsob Pythonu, jak serializovat a deserializovat složité objektové struktury.

- **Výhody**: Dokáže zpracovat téměř jakýkoli objekt Pythonu, včetně instancí vlastních tříd, funkcí a složitého vnořování.
- **Nevýhody**: Výsledný binární formát **není bezpečný** (deserializace dat z nedůvěryhodného zdroje je velkým bezpečnostním rizikem, protože může spustit libovolný kód) a **není kompatibilní** s jinými programovacími jazyky.

Příklad pickle

```python
import pickle

class DataContainer:
    def __init__(self, value):
        self.value = value

data = DataContainer(42)

# 1. Serialization (Writing to File)
with open('data.pkl', 'wb') as f: # 'wb' = write binary
    pickle.dump(data, f)

# 2. Deserialization (Reading from File)
with open('data.pkl', 'rb') as f: # 'rb' = read binary
    reloaded_data = pickle.load(f)

print(f"Reloaded type: {type(reloaded_data)}")
print(f"Reloaded value: {reloaded_data.value}")
# Output: Reloaded value: 42
```

JSON (JavaScript Object Notation)

JSON je dominantní formát pro výměnu dat na webu, především díky své jednoduchosti a přímému mapování na datové struktury běžných programovacích jazyků.

- **Struktura:** Používá **páry klíč-hodnota** (jako Python slovníky) a seřazené seznamy hodnot (jako Python seznamy). Je to člověkem čitelný text.
- **Modul:** Modul `json` se používá k převodu objektů Pythonu (`dict`, `list`, `str`, `int`, `float`, `bool`, `None`) na řetězce JSON a naopak.
- **Výhody: Člověkem čitelný** a univerzálně podporovaný webovými službami a téměř každým programovacím jazykem.
- **Nevýhody:** Může serializovat pouze **standardní primitivní typy** (řetězce, čísla, seznamy, slovníky, booleany, None). Vlastní objekty musí být před serializací převedeny ručně (např. na slovník).

Klíčové operace JSON

- `json.dumps()`
  - Dump String. Serializuje objekt Pythonu do řetězce ve formátu JSON.
  - Python objekt → Řetězec
- `json.loads()`
  - Load String. Deserializuje řetězec ve formátu JSON na objekt Pythonu.
  - Řetězec → Python objekt
- `json.dump()`
  - Dumpuje (zapisuje) objekt Pythonu přímo do otevřeného souborového proudu.
  - Python objekt → Soubor
- `json.load()`
  - Načítá (čte) objekt ve formátu JSON přímo z otevřeného souborového proudu.
  - Soubor → Python objekt

Příklad json

```python
import json

data = {
    "product_id": 500,
    "available": True,
    "tags": ["electronics", "office"]
}

# Serialization to a string
json_string = json.dumps(data, indent=2)
print(json_string)

# Deserialization from a string
reloaded_data = json.loads(json_string)

print(f"Product ID after load: {reloaded_data['product_id']}"
    )
```

Příklad json

```python
import json

data = {
    "name": "Widget",
    "price": 19.99,
    "available": True,
    "details": [1, 2, 3]
}

# 1. Serialization (Writing to File)
with open('data.json', 'w') as f:
    # The 'indent' argument makes the file human-readable
    json.dump(data, f, indent=4)

# 2. Deserialization (Reading from File)
with open('data.json', 'r') as f:
    reloaded_data = json.load(f)

print(f"Reloaded price: {reloaded_data['price']}")
# Output: Reloaded price: 19.99
```

Comma-Separated Values (CSV)

CSV je nejjednodušší a nejběžnější formát pro tabulková data (data organizovaná v řádcích a sloupcích).

- **Struktura:** Každý záznam (řádek) je na novém řádku a pole (sloupce) v záznamu jsou typicky oddělena oddělovačem, nejčastěji čárkou.
- **Modul:** Modul standardní knihovny `csv` zpracovává čtení a zápis CSV souborů a bezpečně řeší oddělovače, které se mohou objevit uvnitř datových polí (např. čárka uvnitř řetězce v uvozovkách).

Klíčové operace `csv`

- `csv.reader`
  - Používá se k iteraci přes řádky v CSV souboru, přičemž každý řádek je zpracován jako **seznam řetězců**.
  - `for row in reader:`
- `csv.writer`
  - Používá se k zápisu dat, bere seznam (řádek) a zapisuje jej do souboru se správnými oddělovači a uvozovkami.
  - `writer.writerow(row_list)`
- `csv.DictReader`
  - Velmi populární. Čte řádky jako slovník, kde klíče jsou **hlavičky sloupců** (z prvního řádku).
  - `for row in DictReader:` (vrací `dict`)

Příklad CSV (čtení s DictReader)

```python
import csv

def read_csv_data(filepath):
    with open(filepath, mode='r', newline='', encoding='utf-8') as file:
        # Use DictReader for easy access by column name
        reader = csv.DictReader(file)
        for row in reader:
            # Access data using column headers (keys)
            print(f"Name: {row['Product']}, Price: {row['Price']}")

# Assumes file.csv contains:
# Product,Price
# Monitor,450
# Keyboard,75
# read_csv_data("data.csv")
```

XML (Extensible Markup Language)

Ačkoli je méně běžný než JSON pro nové webové služby, XML je stále zásadní pro konfigurační soubory, dokumentaci a starší systémy.

- **Struktura:** Používá stromovou strukturu definovanou **tagy** a atributy.
- **Modul:** Modul `xml.etree.ElementTree` je standardní přístup Pythonu pro parsování a navigaci v dokumentech XML.
- **Složitost:** Obecně složitější na parsování než JSON, ale podporuje explicitnější validaci schématu a bohatší datové typování.

Přístup k databázi (Relační databáze)

Interakce Pythonu s relačními databázemi je vysoce standardizovaná a nabízí vrstvy abstrakce od přímého provádění SQL až po objektově orientovanou manipulaci.

Specifikace Python Database API (PEP 249)

Aby byla zajištěna konzistence a přenositelnost, všechny hlavní databázové ovladače pro Python (pro PostgreSQL, MySQL, SQL Server atd.) dodržují specifikaci DB API 2.0.

- **Cíl:** Poskytnout jediné, konzistentní rozhraní, takže kód napsaný pro jeden databázový ovladač lze snadno přizpůsobit pro jiný.
- **Klíčové objekty:** API se točí kolem dvou primárních objektů:
  1. **Connection Object (Objekt připojení):** Představuje aktivní připojení k databázi.
     - **Metody:** `connect()`, `cursor()`, `commit()`, `rollback()`, `close()`.
  2. **Cursor Object (Objekt kurzoru):** Používá se k provádění příkazů SQL a správě kontextu operace načítání (fetch).
     - **Metody:** `execute(sql, parameters)`, `executemany(sql, seq_of_parameters)`, `fetchone()`, `fetchmany()`, `fetchall()`.

Přímý přístup (použití `sqlite3`)

Standardní knihovna obsahuje modul `sqlite3`, který je vynikajícím příkladem ovladače kompatibilního s DB API 2.0. Používá se pro lehké, serverless databáze založené na souborech.
Kroky:

1. **Connect (Připojit):** Navázání připojení (`sqlite3.connect()`).
2. **Cursor (Kurzor):** Vytvoření kurzoru (`connection.cursor()`).
3. **Execute (Provést):** Spuštění parametrizovaného SQL dotazu (`cursor.execute(sql, parameters)`).
4. **Fetch (Načíst):** Získání výsledků (`cursor.fetchone()`, `cursor.fetchall()`).
5. **Commit/Close (Potvrdit/Zavřít):** Potvrzení transakce pro modifikaci dat (`connection.commit()`) a uzavření připojení (`connection.close()`).

Přímý přístup (použití `sqlite3`)

POZOR: Bezpečnostní poznámka: Parametrizované dotazy

Je zásadní používat `?` nebo pojmenované zástupné znaky v řetězci SQL a předávat uživatelská data jako samostatnou n-tici/slovník metodě `.execute()`. Tím se zabrání útokům **SQL Injection**, protože ovladač se postará o bezpečné escapování dat.

Přímý přístup (použití sqlite3) – příklad

```python
import sqlite3

# The 'with' statement ensures the connection is closed automatically
with sqlite3.connect('example.db') as conn:
    cursor = conn.cursor()

    # SAFE: Data passed as a separate parameter tuple
    user_id = 1
    cursor.execute("SELECT name, email FROM users WHERE id = ?", (user_id,))

    result = cursor.fetchone()
    # ... process result
```

Object-Relational Mappers (ORMs)

ORM poskytují vrstvu abstrakce na vysoké úrovni, která mapuje databázové tabulky na **Python třídy** a řádky na **Python objekty**. To umožňuje vývojářům interagovat s databází pomocí objektově orientovaných metod a vlastností namísto psaní raw SQL. **SQLAlchemy:** Dominantní ORM v ekosystému Pythonu. Podporuje dva hlavní režimy:

1. **SQLAlchemy Core:** Funguje jako databázový toolkit, umožňující programové generování SQL dotazů pomocí Python objektů (např. `select(users_table).where(...)`) bez psaní raw SQL řetězců, ale stále se zaměřuje na SQL koncepty.
2. **SQLAlchemy ORM:** Mapuje databázové tabulky na **Deklarativní modely** (Python třídy). To umožňuje vývojářům používat jednoduchý kód Pythonu k dotazování, vkládání a aktualizaci dat.

Klíčové výhody ORM

- **Vysoká produktivita:** Píše se méně opakujícího se SQL kódu.
- **Udržovatelnost:** Změny schématu databáze se často odrážejí v definicích tříd Pythonu, což centralizuje logiku.
- **Přenositelnost:** ORM mohou snadno přepínat mezi různými databázovými backendy (např. SQLite, PostgreSQL, MySQL) bez významných změn kódu.
- **Bezpečnost:** Parametrizovaná konstrukce dotazů je automaticky řešena ORM, což zabraňuje riziku SQL injection.

Koncept SQLAlchemy ORM

```python
# Conceptual Example of an ORM Model
# Represents a database table 'products'
class Product(Base):
    __tablename__ = 'products'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    price = Column(Float)

# Usage (Querying)
# Instead of "SELECT * FROM products WHERE price > 100"
# You write:
# expensive_products = session.query(Product).filter(Product.
    price > 100).all()
```

Ladění a logování

Ladění pomocí výpisů (Dumps)

„Ladění pomocí výpisů“ označuje praxi vkládání explicitních příkazů `print` nebo podobných výstupních příkazů do kódu za účelem kontroly stavu proměnných, toku spuštění a hodnot vrácených funkcemi v konkrétních bodech. Ačkoliv je tato technika jednoduchá, zůstává nejběžnější a nejrychlejší metodou ladění.

Principy efektivního ladění výpisy

1. **Kontextový výstup:** Vždy vypisujte popisný štítek spolu s hodnotou proměnné, abyste jasně identifikovali, který řádek kódu výstup vygeneroval.

```
# Ineffective:
# print(x)

# Effective (Contextual):
print(f"--- LOOP START: i={i}, x={x}")
```

Principy efektivního ladění výpisy

2. **Kontrola stavu:** Výpisy jsou nezbytné pro kontrolu:
   - **Vstup/Výstup:** Potvrzení, že funkce přijímá očekávané argumenty a vrací správný výsledek.
   - **Podmíněná logika:** Ověření, která větev bloku `if`/`elif`/`else` se provádí.
   - **Iterace cyklu:** Sledování stavu proměnných uvnitř cyklu, zejména u měnitelných kolekcí.

Principy efektivního ladění výpisy

3. **Použití `repr()` a `pprint`:**
   - `repr()`: Pro složité nebo vlastní objekty poskytuje tisk `repr(obj)` (nebo použití specifikátoru `!r` v f-strings: `f'obj!r'`) často jasnější a méně nejednoznačnou řetězcovou reprezentaci než `str(obj)`.
   - `pprint` (Pretty Print): Modul `pprint` je neocenitelný pro výpis velkých, hluboce vnořených datových struktur (jako jsou slovníky nebo seznamy). Formátuje výstup úhledně s odsazením a zalomením řádků, takže je čitelný.

```
1 import pprint
2 # A large, nested dictionary
3 data = {'user': {'id': 101, 'name': 'Alice', 'roles': ['admin
      ', 'dev']}}
4
5 # Standard print is hard to read:
6 # print(data)
7
8 # Pretty print is easy to read:
9 pprint.pprint(data)
```

Použití `sys.stderr` pro čistší výstup

Při ladění interaktivních aplikací (např. webový server) může míchání ladicího výstupu s běžným výstupem aplikace způsobit problémy.

- **Standardní praxe:** Použijte `print()` k zápisu ladicích zpráv do **Standardního chybového výstupu** (`sys.stderr`) místo výchozího **Standardního výstupu** (`sys.stdout`).
- **Výhoda:** Tím se oddělí ladicí zprávy od čistého datového výstupu aplikace, což usnadňuje filtrování nebo přesměrování toků.

```python
import sys

def debug_print(*args):
    """Prints debug messages to stderr."""
    print("DEBUG:", *args, file=sys.stderr)

# debug_print("Starting data processing...")
```

Riziko: Zapomenuté výpisy

Hlavní nevýhodou této metody je riziko, že zapomenete odstranit ladicí příkazy `print`, což může zhoršit čitelnost produkčních logů nebo rozhraní aplikace. **Zmírnění:**

- Používejte konzistentní, snadno vyhledatelnou předponu (jako `DEBUG:`, `XXX` nebo specifické emoji), abyste mohli rychle najít a odstranit dočasné příkazy print.
- Lze použít vlastní funkci, ve které lze snadno ladící výpisy vypnout.
- Můžete využít konstanty `DEBUG` a konstrukci `if DEBUG:`
- Pro cokoli složitějšího nebo trvalejšího přejděte od jednoduchých výpisů k dedikovanému **logovacímu frameworku** (probráno v další části).

Standardní logovací knihovna

Modul `logging` je komplexní a vysoce konfigurovatelný framework Pythonu pro správu událostí, které nastanou během běhu softwaru. Poskytuje strukturovanou a škálovatelnou alternativu k jednoduchým příkazům `print()`.

Proč používat Logging místo Print?

Vlastnost	`print()`	Modul `logging`
Cíl	Fixní (stdout/stderr).	Konfigurovatelný (soubory, konzole, síť, databáze).
Kontext	Žádný (vyžaduje ruční popis).	Automatický čas, název modulu, číslo řádku, úroveň logu.
Řízení	Žádné (všechny zprávy se zobrazí).	Jemné řízení pomocí **úrovní** (např. v produkci zobrazit jen chyby).
Výkon	Může výrazně zpomalit I/O.	Vysoce optimalizovaný; vypnuté logy mají minimální režii.

Hlavní komponenty logování

Logovací systém je postaven na čtyřech spolupracujících komponentách:

1. **Loggers (Logery):** Vstupní body do logovacího systému. Jsou to objekty, se kterými kód interaguje pro záznam událostí.
   - *Best Practice:* Získejte objekt logeru pro každý modul, obvykle pomocí `logging.getLogger(__name__)`.

Hlavní komponenty logování

2. **Levels (Úrovně):** Definují závažnost zprávy. Zprávy pod nakonfigurovanou úrovní jsou ignorovány. (V sestupném pořadí závažnosti):
   - `CRITICAL` (50): Pád systému, nelze pokračovat.
   - `ERROR` (40): Vážný problém, provádění pokračuje.
   - `WARNING` (30): Indikace potenciálního problému.
   - `INFO` (20): Potvrzení, že věci fungují podle očekávání.
   - `DEBUG` (10): Podrobné informace, typicky zajímavé pouze při diagnostice problému.

Hlavní komponenty logování

3. **Handlers (Handlery):** Určují, **kam** záznamy logu půjdou.
   - *Příklady:* `StreamHandler` (konzole), `FileHandler` (soubor), `SMTPHandler` (email).
4. **Formatters (Formátovače):** Definují rozvržení a obsah záznamu logu (např. zahrnutí časového razítka, úrovně logu, zprávy).

Základní konfigurace a použití

Jednoduché nastavení se obvykle provádí v hlavním modulu aplikace.

```
1  import logging
2
3  # 1. Configuration (Set up the root logger)
4  logging.basicConfig(
5      level=logging.INFO, # Global minimum level to process
6      format='%(asctime)s - %(name)s - %(levelname)s - %(
           message)s',
7      filename='app.log', # Directs output to a file
8      filemode='a'        # Append mode
9  )
10
11 # 2. Get a Logger (using the module name for context)
12 logger = logging.getLogger(__name__)
```

Základní konfigurace a použití

```
# 3. Logging Messages
logger.debug("This message is ignored because level is INFO.")
logger.info("Application started successfully.")
logger.warning("Configuration file not found, using defaults.")
logger.error("Failed to connect to the database.")

# Special case: Logging Exceptions
try:
    1 / 0
except ZeroDivisionError:
    # log.exception() automatically logs the message AND the full traceback.
    logger.exception("An unhandled exception occurred during division.")
```

Pokročilá konfigurace (hierarchie)

- Logery tvoří **hierarchickou strukturu** (jako složky). Loger s názvem `myapp.database` je potomkem logeru `myapp`.
- **Propagace:** Ve výchozím nastavení záznam logu putuje hierarchií nahoru od zdrojového logeru k jeho rodiči a nakonec ke kořenovému (root) logeru.
- **Centrální řízení:** Tato struktura umožňuje nakonfigurovat obecný loger (např. `myapp`) pro zpracování všech zpráv s úrovní `INFO`, ale nastavit specifický podřízený loger (např. `myapp.performance`) na úroveň `DEBUG` pro selektivní sběr podrobnějších informací.

Debuggery (interaktivní inspekce kódu)

Debugger je softwarový nástroj, který umožňuje programátorovi spouštět program krok za krokem, kontrolovat stav programu (proměnné, paměť, zásobník volání) a upravovat tok provádění za běhu programu. Je to nejúčinnější způsob, jak porozumět složitým cestám provádění a izolovat těžko nalezitelné chyby.

Standardní Python Debugger (pdb)

Standardní knihovna obsahuje Python Debugger (`pdb`), mocný interaktivní nástroj pro příkazovou řádku. **Aktivace:**

- **Manuální Breakpoint:** Vložte `import pdb; pdb.set_trace()` kamkoli do kódu pro pozastavení provádění na tomto konkrétním řádku.
- **Post-Mortem:** Spusťte skript s přepínačem `-m pdb`, nebo použijte `pdb.pm()` poté, co došlo k neošetřené výjimce, abyste vstoupili do debuggeru v místě pádu.
- **Python 3.7+:** Použijte vestavěnou funkci `breakpoint()` místo `import pdb; pdb.set_trace()`.

Standardní Python Debugger (pdb)

Klíčové příkazy pdb:

- `n` (`next`) – Provede aktuální řádek a posune se na další řádek v *aktuální funkci*. **Nevstupuje do volání funkcí.**
- `s` (`step`) – Provede aktuální řádek a posune se. **Vstoupí do volání funkce**, pokud je aktuální řádek voláním funkce.
- `c` (`continue`) – Pokračuje v provádění normálně, dokud není zasažen další breakpoint nebo program neskončí.
- `r` (`return`) – Pokračuje v provádění, dokud se aktuální funkce nevrátí.
- `p výraz` (`print`) – Vyhodnotí a vytiskne hodnotu výrazu (proměnná, volání funkce atd.) v aktuálním rozsahu.
- `l` (`list`) – Vypíše zdrojový kód kolem aktuálního řádku provádění.
- `w` (`where`) – Vytiskne kompletní **stack trace** (zásobník volání).
- `b číslo_řádku` (`break`) – Nastaví nový breakpoint na specifikovaném řádku.
- `q` (`quit`) – Okamžitě ukončí debugger a program.

Debuggery v IDE

Zatímco `pdb` je univerzální, většina profesionálního vývoje v Pythonu spoléhá na lepší rozhraní a funkce debuggerů v IDE (např. ve VS Code, PyCharm nebo jiných IDE). **Funkce:**

- **Grafické Breakpointy:** Nastavení breakpointů kliknutím na okraj editoru.
- **Okna pro sledování proměnných:** Průběžně zobrazují hodnoty všech proměnných v aktuálním rozsahu a libovolné uživatelsky definované výrazy.
- **Vizuální navigace zásobníkem:** Snadné procházení zásobníku volání pro zobrazení stavu proměnných ve volajících funkcích.
- **Podmíněné Breakpointy:** Přeruší provádění pouze v případě, že je splněna určitá podmínka (např. `i > 100`).

Alternativa ipdb

ipdb je knihovna třetí strany, která obaluje pdb, ale integruje se s IPythonem, čímž poskytuje lepší funkce pro inspekci:

- **Doplňování tabulátorem:** Umožňuje automatické doplňování názvů proměnných a příkazů.
- **Zvýrazňování syntaxe:** Zlepšuje čitelnost v terminálu.
- **Lepší Tracebacks:** Často prezentuje jasnější výstup než standardní pdb.

Filozofie ladění

- **Nehádejte, inspektujte:** Hlavním přínosem debuggeru je odklon od hádání stavu programu (jak se dělá u výpisů) k definitivní **inspekci** skutečného stavu v přesném okamžiku selhání.
- **Sledujte chybu:** Použijte debugger ke sledování provádění zpětně od bodu selhání nahoru zásobníkem volání (příkaz `w` nebo vizuální zásobník), abyste identifikovali, kde byla nesprávná hodnota *zavedena*, nejen kde byla použita.

Online komunikace

Webový klient a server (síťová komunikace)

Pochopení modelu Klient-Server je nezbytné pro každou moderní aplikaci, protože tvoří základ internetu, webových aplikací a interakcí s API.

Model Klient-Server

- **Klient:** Program (jako webový prohlížeč nebo skript Pythonu používající knihovnu jako `requests`), který **iniciuje připojení** a požaduje zdroj nebo službu.
- **Server:** Program (jako Apache, Nginx nebo backend aplikace v Pythonu), který **naslouchá připojením** a odpovídá na požadavky klienta poskytnutím dat nebo provedením služby.

Základní síťová abstrakce: Sokety

Na nejnižší úrovni síťového programování je **soket** primárním mechanismem pro komunikaci.

- Soket je jeden koncový bod obousměrného komunikačního spojení mezi dvěma programy běžícími v síti. Umožňuje programům odesílat a přijímat data.
- Vestavěný modul `socket` poskytuje přímý přístup k API soketů Berkeley.
- **Typy:**
  - `SOCK_STREAM` (TCP): Spojově orientovaný. Poskytuje spolehlivé, seřazené doručení s kontrolou chyb. Používá se pro HTTP a většinu aplikačních protokolů.
  - `SOCK_DGRAM` (UDP): Nespojový. Poskytuje rychlejší, ale nespolehlivé doručení. Používá se pro streamování a časově kritická data.

Aplikační vrstva: HTTP

Většina webové komunikace spoléhá na protokol HTTP (HyperText Transfer Protocol). **HTTP klienti v Pythonu:**

- `urllib.request`: Modul standardní knihovny pro načítání URL.
- `requests` (Doporučená knihovna třetí strany): De facto standard pro provádění klientských požadavků. Poskytuje jednoduché, „Pythonic“ rozhraní pro všechny hlavní HTTP metody (`GET`, `POST`, `PUT`, `DELETE` atd.).

Příklad klienta s použitím requests

```python
import requests

# Making a GET request (Client functionality)
response = requests.get('https://api.github.com/users/google')

if response.status_code == 200:
    # Parsing the response body (JSON)
    data = response.json()
    print(f"Name: {data['name']}")
else:
    print(f"Error: {response.status_code}")
```

Webové servery v Pythonu

Python je široce používán k vytváření aplikací na straně serveru (backendů). Ty spoléhají na protokoly definované v PEP pro komunikaci se softwarem webového serveru (jako Nginx).

- **WSGI (Web Server Gateway Interface):** Standard (PEP 3333), který definuje jednoduché, univerzální rozhraní mezi webovými servery (např. Gunicorn, uWSGI) a webovými aplikacemi napsanými v Pythonu (např. Flask, Django). To zajišťuje přenositelnost aplikací mezi různými servery.
- **ASGI (Asynchronous Server Gateway Interface):** Moderní standard (nástupce WSGI) navržený pro asynchronní aplikace. Zvládá více protokolů, včetně HTTP, WebSockets a long-poll spojení, což umožňuje vysokou konkurenci (např. FastAPI, Django Channels).

Příklad serveru (konceptuální Flask/WSGI)

```python
from flask import Flask
app = Flask(__name__)

# This function (view) is executed by the server when a client makes a GET request
@app.route('/hello')
def index():
    return "Hello, World! (Served by Python)"

# if __name__ == '__main__':
#     # In production, this would be run by a WSGI server like Gunicorn
#     app.run()
```

Práce s API (integrace externích dat)

API (Application Programming Interface) definuje sadu pravidel, protokolů a nástrojů, které umožňují různým softwarovým komponentám vzájemně komunikovat. Při práci s externími službami Python obvykle funguje jako **klient**, který konzumuje data poskytovaná API serveru třetí strany.

Základy API

- **Endpointy:** Specifické URL na serveru, které provádějí definované akce (např. `/users`, `/products/101`).
- **HTTP Metody (Slovesa):** Definují akci, která se má provést:
  - `GET`: Načtení dat. (Nejběžnější pro čtení dat).
  - `POST`: Vytvoření nových dat.
  - `PUT/PATCH`: Aktualizace existujících dat.
  - `DELETE`: Odstranění dat.
- **Autentizace:** Proces ověření identity klienta, obvykle pomocí **API klíčů**, **Bearer Tokenů** nebo **OAuth2** (tokeny předávané v hlavičce požadavku).
- **Rate Limiting (Omezení rychlosti):** API ukládají limity na počet požadavků, které může klient provést v určitém časovém rámci, aby chránila svou službu.

Standardní pracovní postup (použití `requests`)

Knihovna `requests` zjednodušuje celý pracovní postup klienta.

1. **Požadavek (Request)** – Sestavení URL a odeslání požadavku s nezbytnými parametry (query, headers, body). Příklad: `response = requests.get(url, params='key': 'value', headers='Authorization': 'Bearer ...')`
2. **Kontrola stavu (Status Check)** – Kontrola stavového kódu HTTP pro ověření úspěchu. Příklad: `if response.status_code == 200:`
3. **Extrakce dat (Data Extraction)** – Parsování těla odpovědi, typicky JSON, do struktury Pythonu (`dict` nebo `list`). Příklad: `data = response.json()`
4. **Zpracování chyb (Error Handling)** – Ošetření stavových kódů jiných než 2xx (např. 404 Not Found, 401 Unauthorized, 429 Rate Limit). Příklad: `response.raise_for_status()` (vyvolá výjimku při špatném stavu)

Práce s parametry a daty

1. **Query parametry** (`GET`): Používají se k filtrování, řazení nebo stránkování výsledků. Předávají se jako slovník do argumentu `params` v `requests`.

```
params = {'limit': 50, 'status': 'active'}
response = requests.get('https://api.example.com/items',
    params=params)
```

2. **Tělo požadavku** (`POST`/`PUT`): Používá se k odesílání dat na server (např. vytvoření nového uživatele). Data jsou obvykle odesílána jako řetězec JSON, což `requests` zpracovává automaticky.

```
new_user_data = {'username': 'JohnDoe', 'email': '
    john@example.com'}
# requests automatically converts the dict to JSON and
    sets the content-type header
response = requests.post('https://api.example.com/users',
     json=new_user_data)
```

Práce se strukturovanými daty z API

Po extrakci dat pomocí `response.json()` získáte standardní objekt Pythonu (`dict` nebo `list`). Výzvou je často validace a převod těchto nestrukturovaných dat na správné, typově bezpečné objekty Pythonu (třídy).

- **Pydantic:** Vysoce doporučená knihovna pro tento úkol. Používá type hinting Pythonu k definování datových schémat a poskytuje automatickou **validaci dat** a serializaci/deserializaci, čímž zajišťuje, že data přijatá z API odpovídají očekáváním.

```python
# Conceptual Example using Pydantic
from pydantic import BaseModel

class UserSchema(BaseModel):
    id: int
    name: str
    email: str

# Assume 'api_data' is a dictionary received from response.
    json()
```

Testování

Testování knihoven a aplikací

Důsledné testování je základním kamenem profesionálního vývoje softwaru. Python má robustní vestavěné nástroje a populární knihovny třetích stran, které zjednodušují tvorbu, spouštění a organizaci testů.

`unittest` (Framework standardní knihovny)

Modul `unittest` (někdy nazývaný **PyUnit**) je nativní framework Pythonu, inspirovaný JUnit. Poskytuje nezbytnou strukturu pro psaní testovacích případů a sad.

- **Klíčové komponenty:**
  1. **Test Fixture (Příprava testu):** Nastavení (např. vytvoření připojení k databázi, příprava dat) a úklid (např. smazání dočasných souborů) potřebné pro spuštění jednoho nebo více testů.
  2. **Test Case (Testovací případ):** Nejmenší testovatelná jednotka. Třída odvozená od `unittest.TestCase` obsahující jednotlivé metody (testy), které provádějí tvrzení (assertions).
  3. **Test Suite (Sada testů):** Kolekce testovacích případů a/nebo jiných testovacích sad.
  4. **Test Runner (Spouštěč testů):** Komponenta, která spouští testy a reportuje výsledky.

`unittest` (Framework standardní knihovny)

- **Klíčové metody v `unittest.TestCase`:**
  1. `setUp()` a `tearDown()`: Metody spouštěné před a po každé testovací metodě. Ideální pro společné nastavení a úklid.
  2. `setUpClass()` a `tearDownClass()`: Metody spouštěné jednou před a po všech testech v celé třídě.
  3. **Metody tvrzení (Assertions):** Metody začínající na `assert`, které kontrolují podmínky (např. `assertEqual(a, b)`, `assertTrue(x)`, `assertRaises(Exception, callable)`).

pytest (vývojářský standard)

Zatímco `unittest` je vestavěný, `pytest` je de facto standardem pro profesionální testování v Pythonu díky své jednoduchosti, mocným funkcím a čitelnosti. **Klíčové výhody oproti** `unittest`**:**

- **Jednoduchost:** Testovací funkce se píší jako jednoduché funkce, nikoli jako metody uvnitř tříd (pokud nejsou potřeba fixtury). Tvrzení používají nativní příkaz Pythonu `assert` (`assert a == b`), nikoli volání metod (`self.assertEqual(a, b)`).
- **Fixtury:** Čistý, modulární způsob správy nastavení testů a závislostí. Fixtury jsou definovány jako funkce pomocí dekorátoru `@pytest.fixture` a jsou vkládány do testovacích funkcí uvedením jejich názvů jako argumentů.
- **Parametrizace:** Snadné spuštění stejné testovací funkce vícekrát s různými sadami vstupních dat, což eliminuje redundantní kód.

Příklad pytest (jednoduchý a čistý)

```python
# Function to test
def add(a, b):
    return a + b

# A simple pytest function (no class inheritance needed)
def test_add_positive():
    # Uses native Python 'assert'
    assert add(1, 2) == 3

def test_add_negative():
    assert add(-1, 1) == 0
```

Příklad pytest (jednoduchý a čistý)

```python
# Example using a fixture (conceptual)
@pytest.fixture
def database_connection():
    # Setup logic here (e.g., connecting to a database)
    conn = connect_db()
    yield conn  # Everything before 'yield' is setup
    conn.close() # Everything after 'yield' is teardown

def test_database_query(database_connection):
    # 'database_connection' is automatically injected by
        pytest
    result = database_connection.query("SELECT 1")
    assert result is not None
```

Mocking a Patchování

Pro unit testování je zásadní izolovat testovaný kód od jeho závislostí (např. databáze, externí API, čas/datum).

- **Mocking** nahrazuje skutečné objekty zástupnými objekty (mocks), které vypadají jako daný objekt, ale vracejí předdefinované výsledky.
- **Nástroje:**
  - `unittest.mock` (Standardní knihovna): Poskytuje třídu `Mock` a funkci `patch`.
  - `MagicMock`: Podtřída `Mock`, která automaticky zpracovává volání speciálních metod (dunder methods).
  - `patch`: Dekorátor nebo kontextový manažer používaný k dočasnému nahrazení objektu (např. externí funkce nebo třídy) mockem během testu.

Příklad Patchování

```python
# Imagine function_under_test calls external_service.fetch_data()

from unittest.mock import patch

@patch('my_module.external_service.fetch_data')
def test_data_processing(mock_fetch_data):
    # 1. Configure the mock to return a predictable result
    mock_fetch_data.return_value = {'status': 'ok', 'data': [10, 20]}

    # 2. Run the function being tested
    result = function_under_test()

    # 3. Assertions: Check result AND check if the dependency was called correctly
    assert result == 30
    mock_fetch_data.assert_called_once_with('default_endpoint')
```

Organizace testů (strukturování sady testů)

Dobře strukturovaná sada testů je nezbytná pro dlouhodobou použitelnost projektu. Zajišťuje, že testy jsou snadno objevitelné spouštěči (jako `pytest`), jasně mapované na zdrojový kód a efektivně prováděné.

Adresářová struktura

Standardní praxí je oddělit testovací kód zcela od kódu aplikace.

- **Samostatný kořenový adresář:** Vytvořte vyhrazený adresář na úrovni projektu pro všechny testovací soubory, typicky nazvaný `tests/`.
- **Odpovídající struktura:** Zrcadlete strukturu vašeho zdrojového kódu uvnitř adresáře `tests/`. Pokud má vaše aplikace modul `src/utils/data_processing.py`, odpovídající testovací soubor by měl být `tests/utils/test_data_processing.py`.
- `__init__.py`: Historicky adresáře potřebovaly soubor `__init__.py`, aby byly rozpoznány jako balíček Pythonu. Ačkoli `pytest` to již striktně nevyžaduje, jeho zahrnutí je stále běžnou praxí pro explicitní strukturu balíčku.

Jmenné konvence (objevitelnost)

Jak `unittest`, tak `pytest` spoléhají na jasné jmenné konvence pro automatické nalezení testů.

- **Soubory testů:** Soubory testů musí být pojmenovány pomocí specifické předpony:
  - `test_*.py` (Nejběžnější a doporučený formát).
  - `*_test.py`
- **Testovací třídy:** Testovací třídy (pokud jsou použity, běžné v `unittest` a pro fixtury v `pytest`) musí začínat na `Test` (např. `class TestUserLogic:`).
- **Testovací funkce/metody:** Testovací funkce nebo metody musí začínat na `test_` (např. `def test_user_creation():`).

Kategorizace a značkování testů

Jak roste množství testů, není nutné neustále spouštět všechny testy. **Test markers (Značky testů)** (pomocí `pytest`) umožňují efektivní, filtrované spouštění.

- Slouží ke kategorizaci testů na základě jejich rozsahu nebo požadavků na spuštění (např. rychlé, pomalé, externí).

1. **Definování značek:** Značky jsou definovány v konfiguračním souboru projektu `pytest.ini` nebo `pyproject.toml`.
2. **Aplikace značek:** Použijte dekorátor `@pytest.mark.<nazev_znacky>` na testovací funkce.

- **Řízení spouštění:** Značky umožňují spustit pouze podmnožinu testů:
  - **Spustit pouze unit testy:** `pytest -m 'not slow'`
  - **Spustit integrační testy:** `pytest -m external`

Kategorizace a značkování testů – příklad

```python
# In tests/test_external.py
import pytest

@pytest.mark.slow
@pytest.mark.external
def test_api_integration():
    # This test takes 10 seconds and hits a real server
    pass
```

Typy testů (podle rozsahu)

Je klíčové organizovat testy na základě jejich rozsahu a rychlosti:

- **Unit Testy (Jednotkové)**
  - Jediná funkce nebo metoda; nejmenší izolovaný kus.
  - **Velmi rychlé**
  - **Žádné závislosti**
- **Integrační Testy**
  - Interakce mezi dvěma nebo více komponentami (např. kód Pythonu a databáze/API).
  - Středně rychlé
  - Vyžadují nastavení databáze/API (často mockované/kontejnerizované).
- **End-to-End (E2E) Testy**
  - Simulace celé uživatelské cesty napříč celým zásobníkem aplikace.
  - Pomalé
  - Vyžadují běžící instanci celé aplikace.

Paralelní zpracování

Vlákna a Procesy (Konkurence vs. Paralelismus)

Paralelního zpracování v Pythonu je dosaženo dvěma hlavními mechanismy: **Vlákny (Threads)** a **Procesy**. Pochopení základního rozdílu mezi nimi – zejména v kontextu **Globálního zámku interpretu (GIL)** – je zásadní pro psaní vysoce výkonného kódu.

- **Proces**
  - Nezávislá instance běžícího programu. Každý proces běží ve svém **vlastním, oddēleném paměťovém prostoru**.
  - Paměť: **oddēlená**.
  - Úlohy vázané na CPU (náročné výpočty), kde je potřeba skutečný **paralelismus**.
- **Vlákno**
  - Sekvence provádění v rámci procesu. Vlákna sdílejí **stejný paměťový prostor** jako rodičovský proces.
  - Paměť: **sdílená**.
  - Úlohy vázané na I/O (čekání na síť/disk), kde je potřeba **konkurence**.

Globální zámek interpretu (GIL)

GIL je mutex (zámek), který brání více nativním vláknům spouštět Python bytecode *současně* v rámci jednoho procesu Pythonu.

- **Dopad na vlákna:** GIL znamená, že i na vícejádrovém stroji podléhají vlákna Pythonu GILu a nemohou dosáhnout **skutečného paralelismu** pro úlohy vázané na CPU. V daném okamžiku může Python bytecode spouštět pouze jedno vlákno.
- **Výhody vláken (navzdory GIL):** Vlákna jsou stále velmi užitečná pro **úlohy vázané na I/O** (např. čekání na odpověď API, čtení souboru). Když vlákno narazí na I/O operaci, **uvolní GIL**, čímž umožní běh jiným vláknům, zatímco první vlákno čeká. To maximalizuje využití CPU překrýváním doby čekání na I/O s dobou výpočtu.

Procesy (skutečný paralelismus)

Jediný způsob, jak ve standardním CPythonu dosáhnout skutečného, vícejádrového paralelismu, je použití více nezávislých procesů.

- **Modul Pythonu:** Modul `multiprocessing`.
- **Mechanismus:** Operační systém se stará o plánování procesů, a protože každý proces má svůj **vlastní interpret Pythonu** a svůj **vlastní GIL**, mohou procesy spouštět Python bytecode na různých jádrech CPU současně.
- **Režie:** Vytváření a správa procesů je výrazně **nákladnější** (vyšší využití paměti a pomalejší start) než vytváření vláken kvůli nutnosti duplikovat celý paměťový prostor.

Procesy (skutečný paralelismus)

- **Meziprocesová komunikace (IPC):** Protože je paměť oddělená, procesy musí používat specifické mechanismy pro komunikaci a sdílení dat:
  - **Roury (Pipes):** Jednoduchý, obousměrný komunikační kanál.
  - **Fronty (Queues):** Datová struktura bezpečná pro vlákna i procesy používaná pro předávání zpráv mezi workery.
  - **Sdílená paměť:** Specializované objekty, které umožňují sdílet data mezi procesy bez jejich kompletní serializace/deserializace.

Knihovny pro konkurenci

- **I/O-Bound** (např. web scraping) → **Vlákna**: `threading` nebo `concurrent.futures.ThreadPoolExecutor`
  - Nízká režie; GIL je uvolněn během čekání na I/O.
- **CPU-Bound** (např. těžká matematika) → **Procesy**: `multiprocessing` nebo `concurrent.futures.ProcessPoolExecutor`
  - Obchází GIL pro dosažení skutečně paralelních výpočtů.

POZNÁMKA: Klíčové poučení

Výběr mezi vlákny a procesy závisí zcela na povaze úkolu: použijte **Procesy pro matematiku** a **Vlákna pro čekání**.

Komunikace a synchronizace

Když běží více vláken nebo procesů současně, jsou potřeba mechanismy, které jim umožní výměnu dat (**Komunikace**) a řízení přístupu ke sdíleným zdrojům (**Synchronizace**), aby se zabránilo poškození dat a souběhům (race conditions).

- **Zámky (`Lock`/`RLock`):** Nejjednodušší synchronizační primitivum.
- **Semafory (`Semaphore`):** Pokročilejší zámek, který řídí přístup k fondu konečných zdrojů.
- **Události (`Event`):** Signalizační mechanismus používaný k upozornění více workerů na konkrétní událost.

Zámky (Lock/RLock)

Nejjednodušší synchronizační primitivum.

- Zámek má dva stavy: zamčeno a odemčeno. Vlákno/proces musí získat zámek před přístupem ke zdroji a poté jej musí uvolnit. Pokud je zámek již držen, žadatel čeká (blokuje), dokud není uvolněn.
- `RLock` **(Reentrant Lock):** Umožňuje stejnému vláknu získat zámek vícekrát, aniž by se zablokovalo. Nezbytné, když funkce, která již drží zámek, volá jinou funkci, která se také pokouší získat stejný zámek.

Semafory (`Semaphore`)

Pokročilejší zámek, který řídí přístup k fondu konečných zdrojů.

- Semafor je inicializován s počítadlem (jeho hodnota). Vlákna/procesy získávají semafor snížením počítadla a uvolňují jej zvýšením počítadla. Pokud je počítadlo na nule, všechny pokusy o získání semaforu blokují, dokud jej jiná entita neuvolní.
- **Případ užití:** Omezení počtu souběžných připojení k databázi nebo serverovému endpointu.

Události (`Event`)

Signalizační mechanismus používaný k upozornění více workerů na konkrétní událost.

- Objekt události spravuje interní příznak (flag). Vlákno může čekat na událost (`event.wait()`), což blokuje, dokud není příznak pravdivý. Jiné vlákno nastaví příznak (`event.set()`), čímž uvolní všechna čekající vlákna. Příznak lze resetovat (`event.clear()`) pro blokování budoucích čekatelů.
- **Případ užití:** Koordinace startovacích rutin nebo upozornění worker vláken, že všechny inicializační kroky jsou dokončeny.

Meziprocesová komunikace (IPC)

Procesy nesdílejí paměť, takže pro bezpečnou výměnu dat jsou vyžadovány specializované techniky. Tyto nástroje jsou často také **thread-safe** (což znamená, že obsahují interní zámky).

- **Fronty (`Queue` / `multiprocessing.Queue`):** Preferovaná, robustní metoda pro komunikaci.
- **Roury (`multiprocessing.Pipe`):** Jednodušší, rychlejší mechanismus pro komunikaci mezi **dvěma** procesy (duplexní, obousměrná nebo jednosměrná).
- **Sdílená paměť:** Používá se, když je potřeba rychle sdílet velké množství dat bez režie plné serializace.

Fronty (`Queue`/`multiprocessing.Queue`)

Preferovaná, robustní metoda pro komunikaci.

- Fronta je sdílená datová struktura, která umožňuje producentům (zapisovatelům) a konzumentům (čtenářům) bezpečně si vyměňovat zprávy. Je navržena tak, aby byla bezpečná pro vlákna/procesy, a řeší nezbytnou serializaci dat (pickling) a deserializaci.
- **Model Producent-Konzument** – jeden proces vkládá data do fronty a jiný je z ní vybírá.

Roury (`multiprocessing.Pipe`)

Jednodušší, rychlejší mechanismus pro komunikaci mezi dvěma procesy (duplexní, obousměrná nebo jednosměrná).

- Roura vrací pár objektů připojení, jeden pro odesílání (`send()`) a jeden pro přijímání (`recv()`).

Sdílená paměť

Používá se, když je potřeba rychle sdílet velké množství dat bez režie plné serializace.

- Vytváří úsek paměti, který je mapován do adresního prostoru více procesů.
- **Nevýhoda:** Přístup ke sdílené paměti musí být ručně synchronizován pomocí explicitního zámku, aby se zabránilo poškození dat.

Knihovna `concurrent.futures`

Tato vysokoúrovňová knihovna zjednodušuje konkurenční programování tím, že automaticky spravuje pooly vláken nebo procesů.

- `ThreadPoolExecutor`: Spravuje pool worker vláken (nejlepší pro úlohy vázané na I/O).
- `ProcessPoolExecutor`: Spravuje pool worker procesů (nejlepší pro úlohy vázané na CPU).
- **Klíčová vlastnost (`submit` a `map`):** Umožňuje odeslat funkce do poolu a vrací objekty `Future`, což jsou zástupné symboly pro výsledek, který bude k dispozici později. Tím se odděluje zadání úlohy od získání výsledku.

Výkonnostní omezení Pythonu

Ačkoli je vývoj v Pythonu rychlý, jeho standardní implementace, **CPython**, často trpí výkonnostními úzkými hrdly ve srovnání s kompilovanými jazyky jako C/C++ nebo vysoce optimalizovanými JIT jazyky jako Java nebo Go.

- **Globální zámek interpretu (GIL)**
- **Dynamické typování a interpretace**
- **Vysokoúrovňové abstrakce a režie objektů**
- **Strategie pro překonání omezení**

Globální zámek interpretu (GIL)

GIL je největším výkonnostním omezením pro úlohy **vázané na CPU** ve standardním Pythonu.

- GIL je mutex, který chrání přístup k objektům Pythonu a brání více vláknům spouštět Python bytecodes současně v rámci jednoho procesu.
- **Dopad:** I na vícejádrových procesorech nemůže modul `threading` Pythonu dosáhnout skutečného paralelního provádění pro výpočetně náročný kód. Omezuje paralelní úlohy na použití více **procesů** (přes `multiprocessing`), což přináší vyšší režii.
- **Zmírnění:** Vliv GILu je zanedbatelný nebo dokonce prospěšný pro úlohy **vázané na I/O**, protože vlákna uvolňují GIL během čekání na externí operace (síť, disk).

Dynamické typování a interpretace

Python je **interpretovaný** jazyk s **dynamickým typováním**, což vyžaduje neustálou režii za běhu.

- **Dynamické typování:** Typy proměnných jsou kontrolovány za běhu, nikoli při kompilaci. To znamená, že pokaždé, když je provedena operace (např. `a + b`), interpret musí zkontrolovat typy `a` a `b` a provést vyhledání ve slovníku, aby našel správnou metodu k provedení (`__add__`). Toto vyhledávání je pomalejší než přímé volání funkce ve staticky typovaných jazycích.
- **Interpretace:** Kód Pythonu je kompilován do přechodné formy (bytecode), která je poté prováděna řádek po řádku virtuálním strojem CPythonu. Tato vrstva provádění je přirozeně pomalejší než spouštění strojového kódu přímo.

Vysokoúrovňové abstrakce a režie objektů

Snadnost použití Pythonu je vykoupena složitostí na pozadí.

- **Všechno je objekt:** Každý kus dat (i celé číslo nebo jednoduchý float) je instancí třídy, uloženou jako struktura C v paměti. Tato struktura obsahuje hodnotu, počet referencí a ukazatel na typ objektu.
- **Paměťová náročnost:** Standardní celé číslo v Pythonu může spotřebovat 28 bytů nebo více paměti, zatímco C integer používá pouze 4 byty. Tato režie paměti může ovlivnit přístup k datům a rychlost, zejména při práci s velkými poli nebo seznamy primitivních typů.

Strategie pro překonání omezení

Pro dosažení vysokého výkonu vývojáři Pythonu obvykle přesouvají pracovní zátěž z pomalého interpretu Pythonu do rychlejšího, optimalizovaného kódu napsaného v C nebo jiných jazycích.

- **C Rozšíření (NumPy/Pandas):** Nejúčinnější strategie. Knihovny jako NumPy a Pandas zpracovávají masivní pole a složité výpočty jejich prováděním ve vysoce optimalizovaném, předkompilovaném **C kódu**, čímž během výpočtu zcela uvolní GIL.
- **JIT Kompilátory (Just-In-Time):**
  - **PyPy:** Alternativní interpret Pythonu, který zahrnuje JIT kompilátor. PyPy dynamicky kompiluje často spouštěný kód Pythonu do strojového kódu za běhu, čímž často dosahuje výrazného zrychlení, zejména u dlouho běžících úloh vázaných na CPU.

Strategie pro překonání omezení

- **Cython:** Jazyk (nadmnožina Pythonu), který umožňuje psát kód se statickými deklaracemi typů. Cython kód je poté přeložen do vysoce optimalizovaného C kódu a zkompilován, což vede k nativnímu C výkonu při zachování syntaxe Pythonu.

Asynchronní Programování (`asyncio`)

Moderní strategie pro překonání omezení vláken u masivních I/O operací.

- **Koncept:** Umožňuje zpracovávat tisíce souběžných připojení v **jediném vlákně** pomocí mechanismu Event Loop (Smyčka událostí).
- **Async/Await:** Klíčová slova používaná k definování korutin a předání řízení zpět smyčce událostí během čekání na I/O.
- **Výhoda:** Eliminuje režii spojenou s přepínáním kontextu operačního systému u vláken. Ideální pro síťové servery a klienty.

Pokročilá témata

Generování kódu (Metaprogramování)

Generování kódu, často součást **Metaprogramování**, je praxe, kdy program píše nebo manipuluje s jinými programy (nebo sám se sebou). V Pythonu to zahrnuje vytváření nebo modifikaci kódových struktur dynamicky za běhu nebo v době kompilace.

Šablonovací systémy (základní přístup)

Nejjednodušší forma generování kódu zahrnuje použití šablonovacího systému ke generování opakujícího se textu nebo kódových souborů na základě vstupních dat.

- Vývojář vytvoří zdrojový soubor obsahující zástupné symboly (šablony). Program poté přečte šablonu, vyplní zástupné symboly dynamickými hodnotami a vypíše výsledný kódový soubor.
- **Případ užití:** Generování konfiguračních souborů, dokumentace API nebo jednoduchých SQL dotazů.
- **Nástroj v Pythonu: Jinja2** (široce používaný ve Flasku a komplexních konfiguracích).

Dynamické spouštění kódu

Python umožňuje konstruovat kód jako řetězec a dynamicky jej spustit za běhu. To poskytuje obrovskou flexibilitu, ale mělo by se používat opatrně kvůli potenciálním problémům s bezpečností a čitelností.

- `exec()`: Spustí řetězec kódu Pythonu jako příkaz.
- `eval()`: Vyhodnotí řetězec kódu Pythonu jako výraz a vrátí výsledek.
- `compile()`: Zkompiluje řetězec do objektu kódu, který lze poté spustit pomocí `exec()` nebo `eval()`. Používá se v případě, kdy je potřeba stejný řetězec spustit vícekrát, protože kompilace se provede pouze jednou.

```
1 code_string = "x = 10\nprint(x * 2)"
2 exec(code_string) # Output: 20
3
4 expression_string = "2 + 2 * 3"
5 result = eval(expression_string) # result = 8
```

Funkce `type()` a Metatřídy

Nejmocnějším způsobem generování nebo úpravy kódových struktur (konkrétně tříd) je použití funkcí objektového modelu Pythonu.

- **Dynamické vytváření tříd pomocí `type()`:** Vestavěnou funkci `type()` lze použít k dynamickému vytvoření třídy za běhu.
- **Metatřídy:** Metatřída je **třída třídy**. Řídí vytváření třídy.

Dynamické vytváření tříd pomocí `type()`

Vestavěnou funkci `type()` lze použít k dynamickému vytvoření třídy za běhu.

- Normálně definujeme třídu pomocí syntaxe `class ClassName(BaseClasses): ...`.
- Dynamicky se třída vytvoří voláním `type(name, bases, dict)`:
  - `name`: Název třídy (řetězec).
  - `bases`: N-tice bázových tříd (rodičů).
  - `dict`: Slovník jmenného prostoru třídy obsahující atributy a metody.

```
1 # Creates a class named 'Point' inheriting from 'object'
2 # with one method 'get_x'
3 def get_x(self): return self.x
4
5 Point = type('Point', (object,), {'x': 0, 'get_x': get_x})
6 p = Point()
7 # print(p.get_x()) # Output: 0
```

Metatřídy

Metatřída je **třída třídy**. Řídí vytváření třídy.

- Když Python vidí definici `class MyClass(...)`, požádá metatřídu (atribut `__metaclass__` nebo výchozí `type`), aby sestavila objekt třídy.
- **Případ užití:** Automatická registrace tříd, aplikace mixinů nebo **vynucování standardů API** napříč více třídami (např. zajištění, že každá třída definuje specifickou metodu). Metatřídy jsou často používány ORM (jako modely v Djangu) k překladu jednoduchých definic tříd do komplexních runtime objektů, které spravují mapování databáze.

Shrnutí

POZNÁMKA:

Zatímco jednoduché spouštění řetězců je flexibilní, použití dynamického vytváření tříd pomocí `type()` a **Metatříd** poskytuje strukturovanou, předvídatelnou kontrolu potřebnou pro robustní generování kódu na úrovni aplikace.

Monkey Patching

Monkey Patching je technika, kdy kód modifikuje nebo rozšiřuje jiný kus kódu (jako třídu, modul nebo funkci) za **běhu**. Zahrnuje nahrazování atributů (metod nebo proměnných) za chodu, typicky pro nahrazení funkcionality třetí strany nebo standardní knihovny bez změny původního zdrojového kódu. Termín „monkey patch“ je odvozen od myšlenky, že kód je „záplatován“ neohrabaným nebo „opičím“ způsobem, často používaným k opravě chyby nebo přidání funkcionality urgentním, lokalizovaným způsobem.

Mechanismus

Python to umožňuje díky své vysoce dynamické povaze: vše je měnitelné a jména jsou vázána za běhu. Atributy jakéhokoli objektu jsou přístupné a lze je nahrazovat dynamicky pomocí tečkové notace (`.`) nebo vestavěných funkcí `getattr()`, `setattr()` a `delattr()`. Proces se skládá ze tří kroků:

1. Definujte **novou funkci** (nebo třídu/metodu) se signaturou, kterou chcete nahradit.
2. Získejte referenci na **původní funkci/metodu**, kterou chcete nahradit.
3. Přiřaďte novou funkci přímo ke jménu staré funkce v jmenném prostoru modulu nebo třídy.

Příklad Monkey Patching

Představte si modul knihovny třetí strany nazvaný `external_api`, který má funkci `fetch_data()`, kterou chceme dočasně nahradit pro test nebo lokální opravu.

```python
# 1. Original (External) Module Code
# file: external_api.py
def fetch_data(url):
    # This is the original, slow network code
    print(f"Fetching data from {url}...")
    return {"status": 500}
```

Příklad Monkey Patching

```python
# 2. Patching Code (The Monkey Patch)
def mock_fetch_data(url):
    """A replacement function that bypasses the network."""
    print("--- Using Monkey Patch Mock ---")
    return {"status": 200, "data": "Local Mock Data"}

import external_api

# Apply the patch: Replace the function in the module's
    namespace
external_api.fetch_data = mock_fetch_data

# 3. Running the patched code
result = external_api.fetch_data("http://example.com")
print(result)

# Output:
# --- Using Monkey Patch Mock ---
# {'status': 200, 'data': 'Local Mock Data'}
```

Běžné případy užití

- **Testování (Mocking):** Nejčastější etické použití. Monkey patching se často používá k nahrazení drahých nebo nekontrolovatelných závislostí (jako síťová volání, přístup k databázi nebo složité interní funkce) jednoduchými **Mocky**, které vracejí předvídatelné hodnoty během unit testování. (Dekorátor `unittest.mock.patch` tento proces čistě automatizuje).
- **Hotfixing:** Aplikace okamžité, lokalizované opravy chyby v závislosti bez čekání na vydání nové verze od správců.
- **Rozšiřování funkcionality:** Přidání nové metody do třídy, kterou nelze přímo modifikovat (např. přidání vlastní metody `to_json` do třídy vestavěné knihovny).

Rizika a nevýhody

- **Čitelnost:** Záplaty (patches) nejsou zřejmé a porušují princip, že „kód by měl dělat to, co se tváří, že dělá“. Ztěžují ladění kódu.
- **Noční můra údržby:** Pokud základní knihovna změní svou interní strukturu nebo signaturu funkce, monkey patch se tiše nebo nepředvídatelně rozbije.
- **Pořadí spouštění:** Pokud se dvě oddělené části aplikace pokusí aplikovat různé monkey patche na stejnou funkci, jeden patch přepíše druhý, což vede k nejasným chybám.
- **Globální dopad:** Monkey patch ovlivňuje **všechna** následná použití opatchovaného kódu v celém běžícím procesu.

Závěr

POZNÁMKA:

Monkey patching je mocný nástroj a je nezbytný pro mockování při testování, ale v běžném aplikačním kódu by se mu mělo vyhýbat ve prospěch standardních OOP technik jako **Dědičnost** a **Kompozice**, pokud neexistuje jiná alternativa.

Hodnocení

Budeme velmi rádi, pokud nám poskytnete hodnocení tohoto kurzu.

VSTUP PRO STUDENTY

Kód kurzu PYTH2

Jméno

Příjmení

Odeslat

ictPRO

Děkuji za pozornost!