Typové informace v Pythonu

- Pavel Tišnovský
- kurzy.python@centrum.cz

Postupné rozšiřování možností Pythonu

- Sémantika
- (Syntaxe)

Nová syntaxe+sémantika v Pythonu 3.x

```
Python 3.5    typové informace
Python 3.6    f-řetězce, async-IO
Python 3.7    klíčová slova async a await
Python 3.8    mroží operátor, poziční parametry
Python 3.9    generické typy
Python 3.10   pattern matching
Python 3.11   skupiny výjimek
Python 3.12   klíčové slovo type + sémantika
```

Deklarace datových typů

- Přidáváno postupně
- PEP 484 - Type Hints a další

Nejpopulárnější jazyky současnosti

Dynamicky typované	Staticky typované
Python	C
JavaScript	C++
Ruby	Go
Perl	Rust
Matlab	Java
PHP	Scala

Přednosti dynamicky typovaných jazyků

- Rychlý cyklus vývoje
  - edit-(compile)-run
- Velmi snadné pro začátečníky
- Ideální pro skriptování
  - CLI
  - skripty na webových stránkách

Zápory dynamicky typovaných jazyků

- Zaručení korektnosti rozsáhlých projektů
- Většinou se vyžaduje větší množství jednotkových testů
  - code coverage není dobrou metrikou!
- Informace o typech se někdy zapisují do komentářů
- IDE nemusí vždy nabízet správné funkce/metody/opravy

To nejlepší z obou světů?

- Volitelné typy

Jazyk	Technologie pro statické typy
JavaScript	TypeScript, Flow
Python	Mypy, Pyright, Pyre
Ruby	Sorbet

Volitelné typy a Python

- Python je dynamicky typovaný
  - a nejsou plány to změnit!
- Typy jsou čistě volitelné
  - přidáno do Pythonu 3.5
  - nazvané "type hints"
  - (aby to vývojáře nestrašilo)
- Statické typové kontroly
  - mypy, pyright, pyre

Statická typová kontrola a Mypy

 Mypy logo

---

Základní základy

```
# - specifikace typu globální proměnné
# - přiřazení nové hodnoty do proměnné


x: int = 42

x = 10
```

```
# - specifikace typu globální proměnné
# - přiřazení nové hodnoty nekompatibilního typu do proměnné


x: int = 42

x = "foo"
```

```
# - specifikace typu lokální proměnné
# - přiřazení nové hodnoty kompatibilního typu do proměnné


def funkce() -> int:
    x: int = 42
    return x * 2
```

```
# - specifikace typu lokální proměnné
# - přiřazení nové hodnoty nekompatibilního typu do proměnné


def funkce(param: float) -> int:
    x: int = 1 / param
    return x
```

- Typ `Any` je přidán automaticky

```
# - funkce bez uvedení typových anotací


def add(a, b):
    """Funkce bez typových anotací."""
    return a + b
```

- Proč `Any`?

```
# - funkce bez uvedení typových anotací
# - zavolání této funkce pro různé typy argumentů


def add(a, b):
    """Funkce bez typových anotací."""
    return a + b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add(1.1, 2.2))
print(add(1 + 1j, 2 + 2j))
print(add("foo", "bar"))
print(add([1, 2, 3], [4, 5, 6]))
print(add((1, 2, 3), (4, 5, 6)))
```

Typové anotace

- specifikují se za dvojtečkou

```python
# - funkce s plným uvedením typových anotací


def add(a: int, b: int) -> int:
    """Funkce s typovými anotacemi."""
    return a + b
```

Typové anotace

- využití

```python
# - funkce s uvedením typových informací
# - zavolání této funkce pro různé typy argumentů


def add(a: int, b: int) -> int:
    """Funkce s typovými anotacemi."""
    return a + b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add(1.1, 2.2))
print(add(1 + 1j, 2 + 2j))
print(add("foo", "bar"))
print(add([1, 2, 3], [4, 5, 6]))
print(add((1, 2, 3), (4, 5, 6)))
```

`bool` nebo `int`?

- Viz specifikaci Pythonu!

```python
assert True+True==2
```

`bool` nebo `int`?

```python
# - funkce s uvedením typových informací
# - zavolání této funkce pro argumenty typu bool a int
# - (ekvivalence mezi True a 1 i False a 0)


def add(a: int, b: int) -> int:
    """Funkce s typovými anotacemi."""
    return a + b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add(1, True))
print(add(1, False))
```

`bool` nebo `int`?

```
# - funkce s uvedením typových informací
# - zavolání této funkce pro argumenty typu bool a int


def add(a: bool, b: bool) -> bool:
    """Funkce s typovými anotacemi."""
    return a and b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add(1, True))
print(add(1, False))
print(add(True, False))
```

Výpis typových anotací

- `any`

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


print(add.__annotations__)
```

- explicitní typy

```
def add(a:int, b:int) -> int:
    return a+b


print(add.__annotations__)
```

Typ `Union`

```
# - funkce s uvedením typových informací
# - použití zobecněného typu Union
# - zavolání této funkce pro různé typy argumenů

from typing import Union


def add(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
    """Funkce s typovými anotacemi."""
    return a + b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
```

```
print(add(1.1, 2))
print(add(1.1, 2.2))
```

Typ `Union`

```
# - funkce s uvedením typových informací
# - použití zobecněného typu Union
# - úprava pro Python 3.10 a vyšší
# - zavolání této funkce pro různé typy argumentů


def add(a: int | float, b: int | float) -> int | float:
    """Funkce s typovými anotacemi."""
    return a + b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add(1.1, 2))
print(add(1.1, 2.2))
```

Dekorátor `@overload`

```
# - funkce s uvedením typových informací pro dvě varianty parametrů
# - použití dekorátoru @overload
# - zavolání této funkce pro různé typy argumentů


from typing import Union, overload


@overload
def add(a: int, b: int) -> int:
    ...


@overload
def add(a: str, b: str) -> str:
    ...


def add(a: Union[int, str], b: Union[int, str]) -> Union[int, str]:
    """Funkce s typovými anotacemi."""
    return a + b


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add("foo", "bar"))
print(add(1, "bar"))
print(add("foo", 2))
```

Dekorátor `@overload`

```
# - funkce s uvedením typových informací pro dvě varianty argumentů
# - použití dekorátoru @overload
# - zavolání této funkce pro různé typy argumentů


from typing import Union, overload


@overload
def add(a: int, b: int) -> int:
    ...


@overload
def add(a: str, b: str) -> str:
    ...


def add(a: Union[int, str], b: Union[int, str]) -> Union[int, str]:
    """Funkce s typovými anotacemi."""
    if isinstance(a, int) and isinstance(b, int):
        return a + b
    elif isinstance(a, str) and isinstance(b, str):
        return a + b
    else:
        raise TypeError("Mixing int and str is not supported!")


# zavolání funkce add s argumenty různých typů
print(add(1, 2))
print(add("foo", "bar"))
print(add(1, "bar"))
print(add("foo", 2))
```

Typ `Optional`

```
# - typ Optional
from typing import Optional

x: Optional[int]

x = None
x = 42
```

```
# - typ Optional

x: int | None
```

```
x = None
x = 42
```

Typované n-tice

- nekorektní varianta

```
# - definice n-tice s jediným prvkem
# - kompatibilita s Mypy
# - (nekorektní typová informace)

from typing import Tuple

p: Tuple[int] = (1, 2, 3)
```

```
# - definice n-tice s jediným prvkem
# - vyžaduje novější verzi Pythonu
# - (nekorektní typová informace)

p: tuple[int] = (1, 2, 3)
```

Typované n-tice

- korektní varianta

```
# - definice n-tice s třemi prvky shodného typu
# - kompatibilita s Mypy

from typing import Tuple

p: Tuple[int, int, int] = (1, 2, 3)
```

```
# - definice n-tice s třemi prvky shodného typu
# - vyžaduje novější verzi Pythonu

p: tuple[int, int, int] = (1, 2, 3)
```

Rozdílné typy prvků v n-tici

```
# - definice n-tice se čtyřmi prvky různých typů
# - kompatibilita s Mypy

from typing import Tuple

p: Tuple[int, float, bool, str] = (1, 3.14, True, "Hello")
```

```
# - definice n-tice se čtyřmi prvky různých typů
# - vyžaduje novější verzi Pythonu

p: tuple[int, float, bool, str] = (1, 3.14, True, "Hello")
```

Typované seznamy

```
# - definice seznamu s prvky typu int
# - kompatibilita s Mypy

from typing import List

lst: List[int] = []
```

```
# - definice seznamu s prvky typu int
# - vyžaduje novější verzi Pythonu

lst: list[int] = []
```

Typované seznamy

```
# - definice seznamu s prvky typu int
# - inicializace prvků
# - kompatibilita s Mypy

from typing import List

lst: List[int] = [1, 2, 3]
```

```
# - definice seznamu s prvky typu int
# - inicializace prvků
# - vyžaduje novější verzi Pythonu

lst: list[int] = [1, 2, 3]
```

- bool/int

```
# - definice seznamu s prvky typu int
# - inicializace prvků
# - použití hodnot True a False

from typing import List

lst: List[int] = [1, True, False]
```

```
# - definice seznamu s prvky typu int
# - inicializace prvků
# - použití hodnot True a False
# - vyžaduje novější verzi Pythonu

lst: list[int] = [1, True, False]
```

Seznamy a typ `Union`

```python
# - definice seznamu s prvky typu int a str
# - inicializace prvků

from typing import List, Union

lst: List[Union[int, str]] = [1, "foo", 42, "bar"]
```

```python
# - definice seznamu s prvky typu int a str
# - inicializace prvků
# - úprava pro Python 3.10 a vyšší

lst: list[int | str] = [1, "foo", 42, "bar"]
```

Typované slovníky

- Slovníky v Pythonu

```python
# - definice slovníku
# - všechny prvky mají shodné typy klíčů i hodnot

d = {}

d["foo"] = 1
d["bar"] = 3
d["baz"] = 10

print(d)
```

```python
# - definice slovníku
# - prvky mají rozdílné typy klíčů i hodnot

d = {}

d["foo"] = 1
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

Typované slovníky

```python
# - definice slovníku
# - specifikace typu klíčů i typu hodnot

from typing import Any, Dict

d: Dict[Any, Any] = {}

d["foo"] = 1
```

```
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

```
# - definice slovníku
# - specifikace typu klíčů i typu hodnot

from typing import Dict

d: Dict[str, float] = {}

d["foo"] = 1
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

Slovníky a typ `Union`

```
# - definice slovníku
# - specifikace typu klíčů i typu hodnot
# - u hodnot je použit typ Union

from typing import Dict, Union

d: Dict[str, Union[int, float, str]] = {}

d["foo"] = 1
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

```
# - definice slovníku
# - specifikace typu klíčů i typu hodnot
# - u hodnot je použit typ Union
# - úprava pro Python 3.10

d: dict[str, int | float | str] = {}

d["foo"] = 1
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

Slovníky a typ `Union`

```
# - definice slovníku
# - specifikace typu klíčů i typu hodnot
# - u klíčů i hodnot je použit typ Union

from typing import Dict, Union

d: Dict[Union[int, str], Union[int, float, str]] = {}

d["foo"] = 1
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

```
# - definice slovníku
# - specifikace typu klíčů i typu hodnot
# - u klíčů i hodnot je použit typ Union
# - úprava pro Python 3.10

d: dict[int | str, int | float | str] = {}

d["foo"] = 1
d["bar"] = 3.14
d[10] = 10
d[42] = "answer"

print(d)
```

Slovníky a typ `Optional`

```
# - definice slovníku
# - specifikace typu klíčů i typu hodnot
# - hodnoty mohou nabývat None

from typing import Dict, Optional

d: Dict[str, Optional[float]] = {}

d["foo"] = 1
d["bar"] = 3.14
d["baz"] = None

print(d)
```

Slovníky a typ `Optional`

```python
# - definice slovníku
# - specifikace typu klíčů i typu hodnot
# - hodnoty mohou nabývat None
# - úprava pro Python 3.10

d: dict[str, float | None] = {}

d["foo"] = 1
d["bar"] = 3.14
d["baz"] = None

print(d)
```

Funkce bez návratové hodnoty

```python
# - funkce bez návratové hodnoty s uvedením typových informací
# - zavolání této funkce pro argumenty typu str a int


def message(msg: str) -> None:
    """Funkce s typovými anotacemi."""
    print(f"Zpráva: {msg}")


# zavolání funkce
message("Ahoj")
```

Typy a funkce vyššího řádu

- typ `callable`

```python
# - funkci printIsPositive lze předat jinou funkci
# - parametr "condition" nemá zapsán datový typ


def printIsPositive(x: float, condition) -> None:
    if condition(x):
        print("Positive")
    else:
        print("Negative")


def positiveFloat(x: float) -> bool:
    return x > 0.0


def positiveInt(x: int) -> bool:
    return x > 0
```

```
printIsPositive(4, positiveFloat)
printIsPositive(-0.5, positiveFloat)
```

```
# - funkci printIsPositive lze předat jinou funkci
# - parametr "condition" má zapsán plný datový typ

from typing import Callable


def printIsPositive(x: float, condition: Callable[[float], bool]) -> None:
    if condition(x):
        print("Positive")
    else:
        print("Negative")


def positiveFloat(x: float) -> bool:
    return x > 0.0


printIsPositive(4, positiveFloat)
printIsPositive(-0.5, positiveFloat)
```

Typy a funkce vyššího řádu

- problém variance

```
# - funkci printIsPositive lze předat jinou funkci
# - parametr "condition" má zapsán datový typ
# - testování typu variance

from typing import Callable


def printIsPositive(x: float, condition: Callable[[float], bool]) -> None:
    if condition(x):
        print("Positive")
    else:
        print("Negative")


def positiveFloat(x: float) -> bool:
    return x > 0.0


def positiveInt(x: int) -> bool:
    return x > 0


printIsPositive(4, positiveFloat)
printIsPositive(-0.5, positiveFloat)
```

```
printIsPositive(1, positiveInt)
printIsPositive(1, positiveInt)
```

Datový typ `range`

```
# - použití datového typu range
# - typ definován pro návratovou hodnotu funkce


def funkce(from_val: int, to_val: int) -> range:
    return range(from_val, to_val)
```

```
# - použití datového typu range
# - typ definován pro parametr funkce


def suma(x: range) -> int:
    return sum(x)


print(suma(range(100)))
```

Problém s variancí

- Týká se podtypů a nadřazených typů
  - v OOP běžné
- Čtyři možné typy variance
  - kovariance
  - kontravariance
  - invariance
  - bivariance

Příklad variancí

- `Jablko` je podtypem typu `Ovoce` ve všech dalších případech

Příklad variancí

- Covariance
  - `List[Apple]` je podtypem `List[Fruit]`
- Contravariance
  - `List[Fruit]` je podtypem `List[Apple]`
- Invariance
  - `List[Fruit]` nemá žádný vztah k `List[Apple]`
- Bivariance
  - `List[Apple]` je podtypem `List[Fruit]`
  - a současně (!!!):
  - `List[Fruit]` je podtypem `List[Apple]`

Proč se o varianci vůbec starat?

- Úzce souvisí s typovým systémem
- A s tím, jaké kontroly lze provést staticky

```
class Fruit {
}

class Orange extends Fruit {
    public String toString() {
        return "Orange";
    }
}

class Apple extends Fruit {
    public String toString() {
        return "Apple";
    }
}

public class Variance1 {
    public static void mix(Fruit[] punnet) {
        punnet[0] = new Orange();
        punnet[1] = new Apple();
    }

    public static void main(String[] args) {
        Fruit[] punnet = new Fruit[2];
        mix(punnet);

        for (Fruit Fruit:punnet) {
            System.out.println(Fruit);
        }
    }
}
```

Statická kontrola typů ok, pád v runtime!

```
class Fruit {
}

class Orange extends Fruit {
    public String toString() {
        return "Orange";
    }
}

class Apple extends Fruit {
    public String toString() {
        return "Apple";
    }
```

```
}

public class Variance2 {
    public static void mix(Fruit[] punnet) {
        punnet[0] = new Orange();
        punnet[1] = new Apple();
    }

    public static void main(String[] args) {
        Fruit[] punnet = new Orange[2];
        mix(punnet);

        for (Fruit Fruit:punnet) {
            System.out.println(Fruit);
        }
    }
}
```

Míchání hrušek s jablky v1

```
# - hierarchie tříd Ovoce <- Hruska a Ovoce <- Jablko
# - funkce `smichej` akceptuje seznam s ovocem
# - přidá do tohoto seznamu Hrusku a Jablko
# - funkci `smichej` voláme s prázdným seznamem pro Ovoce

from typing import List


class Ovoce:
    """Třída, která je předkem tříd Hruska i Jablko."""

    pass


class Hruska(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Hruska"


class Jablko(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Jablko"


def smichej(kosik: List[Ovoce]) -> None:
```

```
    """Do košíku se přidá jedna hruška a jedno jablko."""
    kosik.append(Hruska())
    kosik.append(Jablko())


# košík, který může obsahovat hrušky i jablka
kosik: List[Ovoce] = []

smichej(kosik)

for ovoce in kosik:
    print(ovoce)
```

Míchání hrušek s jablky v2

```
# - hierarchie tříd Ovoce <- Hruska a Ovoce <- Jablko
# - funkce `smichej` akceptuje seznam s ovocem
# - přidá do tohoto seznamu Hrusku a Jablko
# - funkci `smichej` voláme s prázdným seznamem pro Hrušky

from typing import List


class Ovoce:
    """Třída, která je předkem tříd Hruska i Jablko."""

    pass


class Hruska(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Hruska"


class Jablko(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Jablko"


def smichej(kosik: List[Ovoce]) -> None:
    """Do košíku se přidá jedna hruška a jedno jablko."""
    kosik.append(Hruska())
    kosik.append(Jablko())
```

```python
# košík, který může obsahovat pouze hrušky
kosik: List[Hruska] = []

smichej(kosik)

for ovoce in kosik:
    print(ovoce)
```

Řešení problému variance v Pythonu

```python
# - hierarchie tříd Ovoce <- Hruska a Ovoce <- Jablko
# - funkce `tiskni` akceptuje seznam s ovocem
# - samotný seznam se přitom ve funkci nemění (jen se čte)
# - funkci `tiskni` voláme s prázdným seznamem pro Hrušky

from typing import List


class Ovoce:
    """Třída, která je předkem tříd Hruska i Jablko."""

    pass


class Hruska(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Hruska"


class Jablko(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Jablko"


def tiskni(kosik: List[Ovoce]) -> None:
    """Vytiskne obsah košíku s ovocem."""
    for ovoce in kosik:
        print(ovoce)


# košík, který může obsahovat pouze hrušky
kosik: List[Hruska] = []

tiskni(kosik)
```

Použití `sequence` a nikoli seznamu

```
# - hierarchie tříd Ovoce <- Hruska a Ovoce <- Jablko
# - funkce `tiskni` akceptuje neměnitelnou sekvenci s ovocem
# - samotný seznam se přitom ve funkci nemění (jen se čte)
# - funkci `tiskni` voláme s prázdným seznamem pro Hrušky

from typing import Sequence


class Ovoce:
    """Třída, která je předkem tříd Hruska i Jablko."""

    pass


class Hruska(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Hruska"


class Jablko(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Jablko"


def tiskni(kosik: Sequence[Ovoce]) -> None:
    """Vytiskne obsah košíku s ovocem."""
    for ovoce in kosik:
        print(ovoce)


# košík, který může obsahovat pouze hrušky
kosik: Sequence[Hruska] = []

tiskni(kosik)
```

Tisk typové anotace

```
# - hierarchie tříd Ovoce <- Hruska a Ovoce <- Jablko
# - funkce `tiskni` akceptuje neměnitelnou sekvenci s ovocem
# - samotný seznam se přitom ve funkci nemění (jen se čte)
# - funkci `tiskni` voláme s prázdným seznamem pro Hrušky
# - tisk anotace funkce `tiskni`
```

```python
from typing import Sequence


class Ovoce:
    """Třída, která je předkem tříd Hruska i Jablko."""

    pass


class Hruska(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Hruska"


class Jablko(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Jablko"


def tiskni(kosik: Sequence[Ovoce]) -> None:
    """Vytiskne obsah košíku s ovocem."""
    for ovoce in kosik:
        print(ovoce)


# košík, který může obsahovat pouze hrušky
kosik: Sequence[Hruska] = []

tiskni(kosik)

# tisk anotace funkce `tiskni`
print(tiskni.__annotations__)
```

Návratové typy jsou kovariantní

```python
# - návratové typy jsou kovariantní

from typing import Callable


class Ovoce:
    """Třída, která je předkem tříd Hruska i Jablko."""
    pass
```

```
class Hruska(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Hruska"


class Jablko(Ovoce):
    """Potomek třídy Ovoce."""

    def __repr__(self) -> str:
        """Tisk 'hodnoty' objektu."""
        return "Jablko"


def utrhni(f: Callable[[], Ovoce]) -> Ovoce:
    """Zavolá funkci, která získá jeden kus ovoce a vrátí ho."""
    ovoce = f()
    return ovoce


print(utrhni(Hruska))
print(utrhni(Jablko))
```

Odkazy

1. PEP 484 -- Type Hints
2. What’s New In Python 3.5
3. 26.1. typing — Support for type hints
4. Type Hints - Guido van Rossum - PyCon 2015 (youtube)
5. Python 3.5 is on its way
6. Type hints

```
def exit():
    .0%0.
```