

Testování, typové anotace, dokumentace

IB111 ZÁKLADY PROGRAMOVÁNÍ

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

10. října 2025

Základní otázky

- Je můj program dobře napsaný?
 - Budu mu rozumět, až jej uvidím po půl roce?
 - Bude mu rozumět někdo jiný?

Základní otázky

- Je můj program dobře napsaný?
 - Budu mu rozumět, až jej uvidím po půl roce?
 - Bude mu rozumět někdo jiný?
- Funguje můj program správně?
 - Dává očekávané odpovědi?
 - Má očekávané chování?
 - Co je vlastně očekávané?

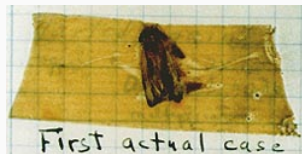
Základní otázky

- Je můj program dobře napsaný?
 - Budu mu rozumět, až jej uvidím po půl roce?
 - Bude mu rozumět někdo jiný?
- Funguje můj program správně?
 - Dává očekávané odpovědi?
 - Má očekávané chování?
 - Co je vlastně očekávané?
- Funguje můj program efektivně?
 - Je dostatečně rychlý?
 - Nevyužívá příliš mnoho paměti?
 - A co jiné zdroje (síťová komunikace, ...)?

Testování

Programy obsahují chyby

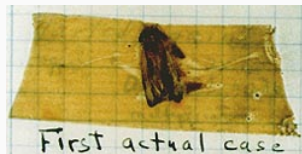
- Chyby jsou běžné,
- všichni chybuji.



Programy obsahují chyby

- Chyby jsou běžné,
- všichni chybuji.

Chyby mohou být závažné



- https://en.wikipedia.org/wiki/List_of_software_bugs

Programy obsahují chyby

- Chyby jsou běžné,
- všichni chybují.



Chyby mohou být závažné



- https://en.wikipedia.org/wiki/List_of_software_bugs

Hledat/odstraňovat chyby je důležité

- ... až tak, že se někdy vypisují odměny za nalezení chyb
https://en.wikipedia.org/wiki/Bug_bounty_program.

Jak odhalit chyby? / Jak ověřit, že program chyby nemá?

- Jeden z důležitých problémů informatiky.
- **Testování.**
 - Různé techniky, různé úrovně (funkce, moduly, ...).
 - Statické i dynamické.
- **Formální metody.**
 - Mohou nejen nalézt chyby, ale i dokázat korektnost.
 - Statická analýza, model checking, deduktivní verifikace.
 - Různé stupně automatizace, škálovatelnosti.
- **Code review člověkem.**
- Nelze dobře realizovat bez **specifikace očekávaného chování.**
 - Jinak max. obecné chyby (přístup mimo rozsah apod.) či slepé „exploratory“ testování: nestačí.
 - Nutno ujasnit si PŘED začátkem testování.

Jak odhalit chyby? / Jak ověřit, že program chyby nemá?

- Jeden z důležitých problémů informatiky.
- **Testování.**
 - Různé techniky, různé úrovně (funkce, moduly, ...).
 - Statické i dynamické.
- **Formální metody.**
 - Mohou nejen nalézt chyby, ale i dokázat korektnost.
 - Statická analýza, model checking, deduktivní verifikace.
 - Různé stupně automatizace, škálovatelnosti.
- **Code review člověkem.**
- Nelze dobře realizovat bez **specifikace očekávaného chování.**
 - Jinak max. obecné chyby (přístup mimo rozsah apod.) či slepé „exploratory“ testování: nestačí.
 - Nutno ujasnit si PŘED začátkem **testování** programování.

(Rozsáhlé téma, jen lehce nakousneme.)

- Velmi důležitá část vývoje software (až 80 % nákladů).
- **Různé úrovně, různé přístupy:**
 - unit testing – testování malých kusů kódu,
 - module testing, integration testing, system testing, ...,
 - coverage-driven, fuzz, combinatorial, performance, static, ...
- **Automatické testování:**
 - kód, jehož cílem je otestovat jiný kód.
- **Co nejvíce různých (druhů) vstupů, resp. chování** (např. u paralelismu):
 - snaha otestovat co nejvíce možností.
- **Testování nikdy není úplné**
 - vstupů je velmi/nekonečně mnoho.



Jedna z možností, jak specifikovat očekávané chování.

Vstupní podmínka (*precondition*) programu, podprogramu, ...

- Jaká data očekáváme:
 - parametry, globální proměnné, externí data (soubory aj.);
 - typ, možné hodnoty, jejich formát, ...

Výstupní podmínka (*postcondition*) programu, podprogramu, ...

- Co bude výsledkem, jaký je efekt:
 - konečnost běhu, návratové hodnoty, vedlejší efekty.

Mohou být zapsány různě.

- Volný jazyk: manuál, dokumentační řetězce v Pythonu atd.
- Základní programové konstrukce: např. assert.
- Rozšířené anotace programů: např. Deal pro Python.
- Formální prostředky – např. Hoareova logika, ACSL pro C,
- Implicitní – název podprogramu, názvy parametrů, ...
- Kombinace.

Nutný obvykle kompromis.

- Vše naprogramováno/formalizováno: drahé sepsání i kontrola.
- Vše implicitní: riziko nepochopení, špatného použití.

Program/podprogram s očekávaným konečným během je korektní vůči dané dvojici vstupní a výstupní podmínky:

- pro všechny vstupy splňující vstupní podmínku skončí a
- efekt odpovídá výstupní podmínce.

Pro vstupy neodpovídající vstupní podmínce není chování (vůbec) definováno!

Program/podprogram s očekávaným konečným během je korektní vůči dané dvojici vstupní a výstupní podmínky:

- pro všechny vstupy splňující vstupní podmínku skončí a
- efekt odpovídá výstupní podmínce.

Pro vstupy neodpovídající vstupní podmínce není chování (vůbec) definováno!

Každý program je korektní vůči nějakým vstupně/výstupním podmínkám: např. false/cokoliv.

Program/podprogram s očekávaným konečným během je korektní vůči dané dvojici vstupní a výstupní podmínky:

- pro všechny vstupy splňující vstupní podmínku skončí a
- efekt odpovídá výstupní podmínce.

Pro vstupy neodpovídající vstupní podmínce není chování (vůbec) definováno!

Každý program je korektní vůči nějakým vstupně/výstupním podmínkám: např. false/cokoliv.

Vývojář musí dodržet zadané vstupně/výstupní podmínky.

- Max zeslabit (rozvolnit) vstupní, zesílit výstupní – může/nemusí být oceněno.
- Příklad: zadání říká, že výstup je seznam, zesílení: bude seřazený.

Příklad – faktoriál.

- Vstup: nezáporné celé číslo `num`.
- Výstup: faktoriál čísla `num`.

Příklad – faktoriál.

- Vstup: nezáporné celé číslo num.
- Výstup: faktoriál čísla num.

```
def fact(num):  
    result = 1  
    while num != 0:  
        result *= num  
        num -= 1  
    return result
```

- Je tento algoritmus korektní?

Příklad – faktoriál.

- Vstup: nezáporné celé číslo `num`.
- Výstup: faktoriál čísla `num`.

```
def fact(num):  
    result = 1  
    while num != 0:  
        result *= num  
        num -= 1  
    return result
```

- Je tento algoritmus korektní? ANO
- A co když zavoláme `fact(-7)`?

Příklad – faktoriál.

- Vstup: nezáporné celé číslo `num`.
- Výstup: faktoriál čísla `num`.

```
def fact(num):  
    result = 1  
    while num != 0:  
        result *= num  
        num -= 1  
    return result
```

- Je tento algoritmus korektní? ANO
- A co když zavoláme `fact(-7)`?
To nás **nezajímá**, `-7` nesplňuje *vstupní podmínku*.

Příklad – největší společný dělitel.

Příklad – největší společný dělitel.

- Vstup: dvě nezáporná celá čísla a , b taková, že alespoň jedno z nich není nula.
- Výstup: největší společný dělitel a i b , podrobněji:

Příklad – největší společný dělitel.

- Vstup: dvě nezáporná celá čísla a , b taková, že alespoň jedno z nich není nula.
- Výstup: největší společný dělitel a i b ,
podrobněji: číslo d takové, že
 - d je dělitelem a i b , tedy $a = xd$ a $b = yd$ pro nějaké celé x, y ,
 - a pokud je z dělitelem a i b , pak $z \leq d$.

Příklad – největší společný dělitel.

- Vstup: dvě nezáporná celá čísla a , b taková, že alespoň jedno z nich není nula.
- Výstup: největší společný dělitel a i b ,
podrobněji: číslo d takové, že
 - d je dělitelem a i b , tedy $a = xd$ a $b = yd$ pro nějaké celé x, y ,
 - a pokud je z dělitelem a i b , pak $z \leq d$.

Poznámka: Kdybychom změnili poslední řádek na „pokud je z dělitelem a i b , pak z dělí d “, pak můžeme na vstupu povolit i dvojici nul.

Příklad – řazení seznamu čísel od nejmenšího po největší.

- Vstup: seznam čísel `my_list`.
- Výstup: seznam čísel `result`, který

Příklad – řazení seznamu čísel od nejmenšího po největší.

- Vstup: seznam čísel `my_list`.
- Výstup: seznam čísel `result`, který
 - je seřazený od nejmenšího po největší, tj. platí
`result[0] <= result[1] <= ... <= result[-1]`

Příklad – řazení seznamu čísel od nejmenšího po největší.

- Vstup: seznam čísel `my_list`.
- Výstup: seznam čísel `result`, který
 - je seřazený od nejmenšího po největší, tj. platí
`result[0] <= result[1] <= ... <= result[-1]`
 - a obsahuje stejná čísla jako vstupní seznam

Příklad – řazení seznamu čísel od nejmenšího po největší.

- Vstup: seznam čísel `my_list`.
- Výstup: seznam čísel `result`, který
 - je seřazený od nejmenšího po největší, tj. platí
`result[0] <= result[1] <= ... <= result[-1]`
 - a obsahuje stejná čísla jako vstupní seznam, a to ve stejném počtu (tj. je to permutace původního seznamu).

- Testování jednotlivých částí kódu (podprogramů).
- Podprogramy, které testují, zda kód funguje správně.

```
def test_fact1():  
    if fact(17) == 355687428096000:  
        print("OK")  
    else:  
        print("Chyba! Funkce fact dává špatný "  
              "výsledek pro vstup 17.")  
    # ...
```

- Výhody oproti „ručnímu“ testování?

- Testování jednotlivých částí kódu (podprogramů).
- Podprogramy, které testují, zda kód funguje správně.

```
def test_fact1():  
    if fact(17) == 355687428096000:  
        print("OK")  
    else:  
        print("Chyba! Funkce fact dává špatný "  
              "výsledek pro vstup 17.")  
    # ...
```

- Výhody oproti „ručnímu“ testování?
 - můžeme spouštět opakovaně,
 - např. po každé změně v kódu.

- Způsob, jak do kódu psát podmínky, jež zaručeně musí platit.
 - Interpret je zkontroluje¹,
 - když neplatí, ukončí program s chybovou hláškou.
- Použití:
 - **testování**,
 - **ověření předpokladu**, jenž nutně *musí platit* v určitém místě programu (často: vstupy podprogramů, hlavička cyklu, ...).

```
def test_fact2():  
    assert fact(17) == 355687428096000, \  
        "Špatný výsledek pro vstup 17."  
    # ...
```

- Nepoužívejte pro kontrolu **uživatelského vstupu**!
 - Selhání příkazu **assert** vždy znamená *chybu v programu*.

¹Nejsou-li zapnuty optimalizace, což standardně (v Pythonu) nejsou.

```
def square_root(x, precision):  
    lower = 0.0  
    upper = x  
    middle = (lower + upper) / 2  
    while abs(middle ** 2 - x) > precision:  
        assert lower ** 2 <= x <= upper ** 2, \  
            "The solution is outside the bounds."  
        if middle ** 2 > x:  
            upper = middle  
        elif middle ** 2 < x:  
            lower = middle  
        middle = (lower + upper) / 2  
    return middle
```


assert pro kontrolu vstupní podmínky

- Užitečné např. v pomocných/knihovních funkcích – pro kontrolu, že nejsou někde volány s nevalidními vstupy.
- (Ne všechny vstupní podmínky se vždy takto zapisují a testují.)

```
def fact(num):  
    assert num >= 0  
    ...
```

Pro účely tohoto předmětu

- V testech nikdy nezkoušíme vstupy, které nesplňují vstupní podmínku zadaných funkcí – je tedy úplně jedno, co se v takovém případě stane.

```
def matrix_mult(left, right):
    l_rows, l_cols = matrix_size(left)
    r_rows, r_cols = matrix_size(right)
    assert l_cols == r_rows, \
        "Incompatible matrix sizes"
    # ...

def matrix_size(matrix):
    rows = len(matrix)
    assert rows > 0, "The matrix is empty"
    cols = len(matrix[0])
    for i in range(1, len(matrix)):
        assert len(matrix[i]) == cols, \
            "Not all rows have the same length"
    return rows, cols
```

```
def argmin(a_list):  
    m = min(a_list)  
    for i, value in enumerate(a_list):  
        if m == value:  
            return i
```

```
assert False
```

- Příkaz `assert False` říká:
„sem se vykonávání programu nemůže dostat“.
- Lepší řešení než „`return` nějaké náhodné hodnoty“; proč?

```
def argmin(a_list):  
    m = min(a_list)  
    for i, value in enumerate(a_list):  
        if m == value:  
            return i
```

`assert False`

- Příkaz `assert False` říká:
„sem se vykonávání programu nemůže dostat“.
- Lepší řešení než „`return` nějaké náhodné hodnoty“; proč?
 - Čitelnost – jasný záměr.
 - Obrana proti chybám – program selže, místo aby počítal s nesmyslnou hodnotou.

Jak a kdy testovat?

Jak

- Začít od jednotlivých, malých částí kódu: podprogramů (včetně pomocných).
 - U složitějších programů mohou být některé podprogramy simulovány.
- Testovat, že chování odpovídá specifikaci:
 - bez ohledu na implementaci testované funkce.
- Zkusit víc různých vstupů:
 - snažit se pokrýt co nejvíc různých možností,
 - nezapomenout na krajní hodnoty.

Kdy

- Tradičně: po každé implementované části.
- Alternativně: předtím, než začnete programovat.
 - Test-Driven Development.
 - Donutí přemýšlet nad tím, co vlastně chcete programovat.

Typové anotace v Pythonu

Silný vs. slabý typový systém – dle míry toho,

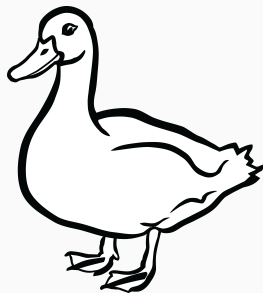
- jak velká omezení typový systém klade,
- jak snadno se ta omezení dají obejít.
- Není zcela jasně definováno, neostrá hranice.

Dynamické vs. statické typování

- **Statické:** proměnné (funkce apod.) mají svůj typ,
 - typy kontrolovány *před spuštěním* programu,
 - typicky v době kompilace,
 - typová chyba zabrání spuštění programu.
- **Dynamické:** objekty mají svůj typ,
 - typy kontrolovány *za běhu* programu,
 - typová chyba ukončí program (nebo vyvolá výjimku apod.).

Dynamické:

- jednodušší, rychlejší vývoj,
- stručnější kód,
- větší flexibilita,
- snazší polymorfismus (duck-typing).



Statické:

- bezpečnější,
- odhalí chyby dříve, než se stanou kritickými,
 - „lépe se Vám v noci spí“,
- často umožňuje rychlejší běh programu,
- informace o typech může mít dokumentační funkci
 - (a kontroluje ji stroj, ne člověk).

- Od verze 3.6 (omezeně i v některých starších verzích).
- Přináší (některé) výhody statického typového systému do dynamicky typovaného jazyka:
 - odhalení potenciálních chyb *před spuštěním* programu,
 - nutnost přemýšlet o typech vede (často) k lepšímu kódu,
 - lepší spolupráce na větších projektech,
 - některá IDE fungují lépe („našeptávání“).

- Od verze 3.6 (omezeně i v některých starších verzích).
- Přináší (některé) výhody statického typového systému do dynamicky typovaného jazyka:
 - odhalení potenciálních chyb *před spuštěním* programu,
 - nutnost přemýšlet o typech vede (často) k lepšímu kódu,
 - lepší spolupráce na větších projektech,
 - některá IDE fungují lépe („našeptávání“).
- Interpret Pythonu je *nekontroluje*.
- Používají je nástroje pro statickou analýzu kódu:
 - součástí některých IDE,
 - samostatné,
 - my budeme používat: *mypy*,
 - <http://mypy-lang.org>.

Typová anotace funkcí

```
def is_triangle(a: int, b: int, c: int) -> bool:
```

- Typy **formálních parametrů** (tj. co funkce očekává).
- Typ **návratové hodnoty** (tj. co funkce slibuje, že vrátí).
- Pokud funkce nic nevrací, používáme **None**.
- Neanotované funkce *mypy* ignoruje (dá se změnit `--strict`).

Typová anotace proměnných

- **Většinou není potřeba** – *mypy* si umí typ odvodit z přiřazení.
- Ale pořád je zde jistá dokumentační vlastnost.

```
city: str = "Sparta"  
soldier_count = 300 # type: int
```

Typová anotace funkcí – část vstupní/výstupní podmínky.

```
def is_triangle(a: int, b: int, c: int) -> bool:
```

- Typy **formálních parametrů** (tj. co funkce očekává).
- Typ **návratové hodnoty** (tj. co funkce slibuje, že vrátí).
- Pokud funkce nic nevrací, používáme **None**.
- Neanotované funkce *mypy* ignoruje (dá se změnit `--strict`).

Typová anotace proměnných

- **Většinou není potřeba** – *mypy* si umí typ odvodit z přiřazení.
- Ale pořád je zde jistá dokumentační vlastnost.

```
city: str = "Sparta"  
soldier_count = 300 # type: int
```

Anotace pro seznamy: `list[typ]`.

- Všechny položky seznamu stejného typu.

Anotace pro ntice: `tuple[typ1, typ2, typ3]` apod.

- Typ pro každou položku ntice.

Anotace pro seznamy: `list[typ]`.

- Všechny položky seznamu stejného typu.

Anotace pro ntice: `tuple[typ1, typ2, typ3]` apod.

- Typ pro každou položku ntice.

```
def average(numbers: list[float]) -> float:  
    return sum(numbers) / len(numbers)
```

```
my_list: list[int] = []
```

```
def minmax(num1: int, num2: int) -> tuple[int, int]:  
    return min(num1, num2), max(num1, num2)
```

Poznámka: Ve starších verzích Pythonu (před 3.10) bylo třeba importovat typové anotace (`List`, `Tuple`) z knihovny `typing`.

Funkce, které někdy vrátí hodnotu a někdy `None`.

- (Ve starším Pythonu pomocí `Optional` z knihovny `typing`.)

```
def safe_divide(p: float, q: float) -> float | None:  
    return None if q == 0 else p / q
```

Funkce, které někdy vrátí hodnotu a někdy `None`.

- (Ve starším Pythonu pomocí `Optional` z knihovny `typing`.)

```
def safe_divide(p: float, q: float) -> float | None:
    return None if q == 0 else p / q
```

Typová synonyma (pro čitelnější anotace).

- Pro jména vlastních typů používáme `CamelCase`.

```
Book = tuple[str, str, float]
```

```
def get_book(database: list[Book], title: str) \
    -> Book | None:
    for book in database:
        book_title, _, _ = book
        if book_title == title:
            return book
```


mypy umí pracovat s `if` a `assert`.

```
result = safe_divide(3.14, 7.0)
```

```
if result is not None:
```

```
    # mypy knows result is not None here
```

```
    print(result + 1)
```

```
if result is None:
```

```
    print("None")
```

```
else:
```

```
    # mypy knows result is not None here
```

```
    print(result + 1)
```

```
assert result is not None
```

```
# mypy knows result is not None here
```

```
print(result + 1)
```

```
list_maybe: list[int | None] = [1, 3, None, 3, 7]
```

Následující s 'mypy' funguje:

```
assert list_maybe[1] is not None  
num = list_maybe[1] + 1
```

```
list_maybe: list[int | None] = [1, 3, None, 3, 7]
```

Následující s 'mypy' funguje:

```
assert list_maybe[1] is not None
num = list_maybe[1] + 1
```

Následující bohužel ne:

- mypy provádí jednoduchou statickou analýzu kódu, nevyhodnocuje plně výrazy.

```
assert list_maybe[i] is not None
num = list_maybe[i] + 1
```

```
list_maybe: list[int | None] = [1, 3, None, 3, 7]
```

Následující s 'mypy' funguje:

```
assert list_maybe[1] is not None
num = list_maybe[1] + 1
```

Následující bohužel ne:

- mypy provádí jednoduchou statickou analýzu kódu, nevyhodnocuje plně výrazy.

```
assert list_maybe[i] is not None
num = list_maybe[i] + 1
```

Řešením je použít novou proměnnou (to je často i čitelnější).

```
ith_elem = list_maybe[i]
assert ith_elem is not None
num = ith_elem + 1
```

```
def maybe_get_pair() -> tuple[int, int] | None:  
    return 0, 0  
  
if maybe_get_pair() is not None:  
    row, col = maybe_get_pair()
```

```
def maybe_get_pair() -> tuple[int, int] | None:  
    return 0, 0
```

```
if maybe_get_pair() is not None:  
    row, col = maybe_get_pair()
```

- mypy se při volání funkcí nedívá „dovnitř“, jen na anotace.
- mypy neví, že dvě *různá* volání funkce vrátí stejný výsledek.
- Bez hlubší analýzy to obecně **nemůžeme** vědět.

```
def fun1() -> int:
    if answer() == 42:
        return answer()
    return 0
```

```
def fun2() -> int:
    result = answer()
    if result == 42:
        return result
    return 0
```

```
def fun1() -> int:
    if answer() == 42:
        return answer()
    return 0
```

```
def fun2() -> int:
    result = answer()
    if result == 42:
        return result
    return 0
```

- Obecně NE!
- Funkce `answer` nemusí být *čistá* a při různých voláních může vracet různé hodnoty:
 - čtení vstupu od uživatele (`input`),
 - náhodná čísla (`random`),
 - globální proměnné,
 - aktuální čas (`datetime`),
 - čtení ze souboru, komunikace po síti, ...


```
def fun1() -> int:
    if answer() == 42:
        return answer()
    return 0
```

```
def fun2() -> int:
    result = answer()
    if result == 42:
        return result
    return 0
```

```
def answer() -> int:
    global_list.append(0)
    return len(global_list)
```

```
global_list: list[int] = []
```

Od 4. kapitoly povinné!

- Anotujte funkce (i pomocné):
 - parametry i návratovou hodnotu.
- Zkuste spustit *mypy* s volbou `--strict`.
- Pokud je všechno v pořádku, jste hotovi.
- Pokud si *mypy* stěžuje, přečtěte si, co říká ...
 - ... a podle toho opravte svůj kód.
 - Ale **ne dodáním nějaké nesmyslné konstrukce**,
 - ani s poznámkou „for mypy to be happy“.
- Nejste-li si jistí, co hlášení od *mypy* znamená,
 - zkuste si ho přečíst (a přeložit z angličtiny),
 - zeptejte se – cvičící(ho), na diskusním fóru, na konzultacích, ...

Proměnné většinou není třeba anotovat.

- Kdyby to bylo nutné, *mypy* si bude stěžovat.

Dokumentace, komentáře

Části kódu s dokumentační funkcí

- píšeme pro sebe i pro ostatní,
- komentáře v kódu,
- dokumentace rozhraní –
 - v Pythonu: dokumentační řetězce (*docstring*),
- názvy (modulů, funkcí, proměnných),
- typové anotace.

Části kódu s dokumentační funkcí

- píšeme pro sebe i pro ostatní,
- komentáře v kódu,
- dokumentace rozhraní –
 - v Pythonu: dokumentační řetězce (*docstring*),
- názvy (modulů, funkcí, proměnných),
- typové anotace.

Nejlepší kód je takový, který se (co nejvíc) dokumentuje sám.

- Smyslem komentářů je usnadnit čtenáři pochopení kódu;
 - komentáře jsou „odpovědi na (očekávané) otázky“.
- Je ovšem vždy lepší zlepšit čitelnost kódu jako takového:
 - vhodně pojmenované proměnné a funkce,
 - vhodná dekompozice funkcí,
 - rozumná struktura programu,
 - ...

Co nedělat?

- Nekomentujte, co a jak kód dělá; komentujte *proč*.
- Nekomentujte špatný kód; přepište jej.
- Neduplikujte informace, které lze snadno vyčíst z kódu.
- Nepoužívejte jiný jazyk než angličtinu.

BAD

```
x += 1  # increment x by one
```

```
a = b   # assign the value of b to a
```

```
direction = 1  # left
```

```
if output:  # print message  
    print(message)
```

```
def set_last_name(user, last_name):  
    # set user's name  
    user.last_name = last_name
```

BAD

```
# iterate over all students in a class
for x in xs:
    # if the student has failed the exam
    if x.score / total < 0.98:
        # set student's grade to F
        x.grade = 5
```


BAD

```
# iterate over all students in a class
for x in xs:
    # if the student has failed the exam
    if x.score / total < 0.98:
        # set student's grade to F
        x.grade = 5
```

Lepší řešení:

```
for student in students:
    if has_failed(student):
        student.grade = GRADE_F
```

```
SOUND_SPEED = 343210  # mm/s
TIMER_PORT = 0x0A  # see datasheet XYZ012
BLINKING_DELAY = 10000  # microseconds

# we cannot use random.choice on a dict,
# see https://bugs.python.org/issue33098

# make sure file will not be processed again
file.completed = True
```

```
def day_of_week(day: int, month: int, year: int) -> int:
    # this uses a variation on Gauss's method as described in
    # en.wikipedia.org/wiki/Determination_of_the_day_of_the_week

    # shift month so that March = 1 and decrease year for Jan, Feb
    if month < 3:
        year -= 1
        month += 10
    else:
        month -= 2

    century = year // 100
    year %= 100

    return (day + (26 * month - 2) // 10 +
            year + (year // 4) +
            (century // 4) - 2 * century) % 7
```

Volně ležící řetězec na začátku funkce (třídy, metody, modulu),

- typicky víceřádkový (uvnitř `""" ... """`).

```
def do_nothing() -> None:
    """This does nothing.

    This really does nothing,
    but does so very efficiently."""
```

- Online nápověda v shellu: `help(do_nothing)` (apod. v IDE),
 - obsahuje i typové anotace funkce.
- K řetězci se dá dostat pomocí `do_nothing.__doc__`.

Konvence pro psaní dokumentace

- Oficiální PEP: <https://www.python.org/dev/peps/pep-0257/>
- Často součást *coding guides*, např. *styl Google*.

Ideální místo pro **popis vstupních a výstupních podmínek**,

- resp. těch jejich částí, na které nestačí typové anotace.

```
def gcd(a: int, b: int) -> int:
    """Returns the greatest common divisor of a, b.

    Assumes that both a, b are nonnegative
    and at most one of them is zero."""
    ...
```

```
def day_of_week(day: int, month: int, year: int) -> int:
    """Returns the day of week (0 for Sunday) for
    the given date in the Gregorian calendar."""
    ...
```

Dokumentace

- Je určena pro ty, kdo budou kód **používat**.
- Vysvětluje, co funkce (modul, program, ...) dělá,
 - např. vstupní a výstupní podmínky.
- Nezabývá se detaily implementace.
- Uživatelský manuál, návod k použití.

Komentáře v kódu

- Jsou určeny pro ty, kdo budou kód **číst**.
 - (To může být i sám autor, o něco později.)
- Vysvětluje, proč kód dělá to, co dělá.