Jak programovat v Pythonu pro středně pokročilé

(B3B33LAR – Laboratoře robotiky)

David Koníček
david.konicek@cvut.cz
david.konicek@email.cz
2024 březen/duben

1

Jak programovat v Pythonu pro středně pokročilé

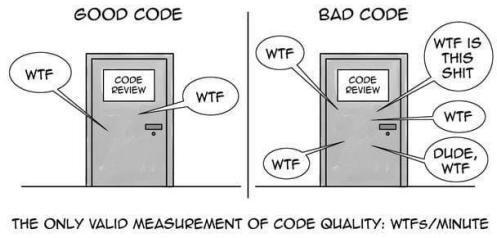
- Úvod
- Python čistý kód
- Python PEP8
- Komentáře
- Dokumentace v kódu
- Identifikátory
- Struktura souboru
- Autorská práva a licence
- Struktura adresářů
- Nástroje
- Využití AI
- Programovací techniky
- Typové kontroly
- GIT – netradiční pohled
- Testování
- Zdroje, další studium

2

Motivace k čistému kódu

4

Jak programovat? Čistě!



THE ONLY VALID MEASUREMENT OF CODE QUALITY: WTFS/MINUTE

5

Motto (3x)

„Program is written once, but read 50 times“
Anonymous

„Programs are meant to be read by humans and only incidentally for computers to execute.“
Donald Knuth

„80% of the lifetime cost of a piece of software goes to maintenance.“
Sun Microsystems: Java Code Conventions (1997)

6

Jaký kód je čistý?

čistý kód	*clean code*
čitelný kód	*readable code*
znovupoužitelný kód	*reusable code*
udržovatelný kód	*maintainable code*
v průmyslové kvalitě	*industry-strength programming*

7

Opak čistého kódu

- Nečitelný, špatně udržovatelný, ….
- „code smell“
- „anti-pattern“

8

Výhody a přínosy čistého kódu

- Snadný k porozumění, **co bude kód dělat**
- Snadný k porozumění i **za několik měsíců / roků**
- Snadný k porozumění pro ostatní **uvnitř týmu**
- Snadný k porozumění **mimo tým** (API, knihovny)
- Snadný/jednodušší pro **testování**

9

Čistý kód je novinka?

- Vždy byla snaha o správný a čitelný kód
- Rozvoj složitosti programů a řešených oblastí ⇨

- Strukturované programování '60
- Modulární programování '70
- Objektové programování '70-'80
- Čistý kód '90-'00
- Návrhové vzory a další paradigmata '00-'20

Čas

10

Programování v 21.století

- **Komplexnost** kódu roste, přesto …
- **Práce v týmu** (vč. předávání „v čase“ a mezi týmy/projekty), proto …

… vyvíjený kód musí být:

- **čitelný**
- **pochopitelný**
- **udržovatelný** (včetně hledání chyb i rozvoje kódu)

11

Kdy programovat čistě?

Vždy!
- business
- univerzity
- open source

Výjimky:
- pokusný a jednorázový kód („trvanlivost“ < 1 den)
- starý kód
- výukové materiály pro promítání (prostorová omezení)
- programátorské soutěže
- hackaton, workshop, …

12

Jak programovat čistě?

- Změna přístupu:
  - jednotlivec
  - tým
  - management
- Standardy a zvyky
- Pravidla a konvence

Co dělat?

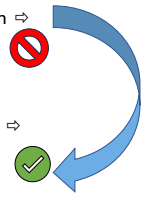
- Sebedisciplína:
  - jednotlivec
  - tým
  - management
- Pravidla programovacího jazyka
- Nástroje statické kontroly
- Nástroje umělé inteligence (AI)

Jak to dělat?

13

Standardy, zvyky, konvence

- Měřítkem "čistoty" je vnímání kódu člověkem ⇨ tedy subjektivní pohled ⇨ **subjektivita**
- Sjednocení pohledu, odstranění subjektivity ⇨ **standardy, zvyky, pravidla a konvence** ⇨ sjednocení pohledu a odstranění duplicity *(good practices)*



14

Univerzálnost pravidel

- Neexistují 100% univerzální pravidla/konvence
- Existují obvykle dodržované pravidla/konvence
- Firmy/projekty/atd. definují vlastní modifikace pravidel
- Dva typické přístupy:
  - **Doktrinářský přístup** = žádné výjimky
  - **Volný přístup** = možné výjimky, nutno ale popsat

15

Čistý kód – (některé) oblasti

- Formátování kódu (*style guide*)
- Identifikátory – smysluplné, dobře popisné
- Komentáře a dokumentace
- Uspořádání souborů a adresářů
- Programátorské praktiky – pochopitelný kód

16

Čistý kód – (některé) oblasti

- Testy – automatizované
- Pokročilá paradigmata (koncepty)
- Zpracování chyby, detekce chyb
- GIT, verzování, týmová práce
- Čistá architektura

17

Programovací jazyky a čistý kód

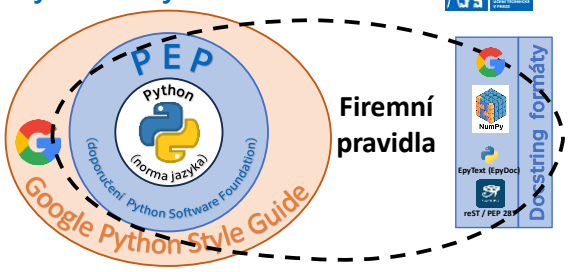
Čistý kód je obecný princip Čistý kód je aplikovatelný na všechny programovací jazyky	
statické	Pascal, C, C++, C#, Java, Ada, TypeScript, Go, …
dynamické	Python, PHP, JavaScript, R, MATLAB, Lisp, Perl, …
databázové	SQL, PL/SQL, T-SQL, CQL (Cassandra), GraphlQL, …
skriptovací	Shell (bash, csh, sh), PowerShell, AWK, SED, CMD …
značkovací	HTML, CSS, XML, …

18

Python – čistý kód

19

Python čistý kód – oblasti



20

Python – základy čistého kódu

- **PEP 8** - Style Guide for Python Code
- **PEP 484** - Type Hints + **PEP 485** - The Theory of Type Hints
- **PEP 257** - Docstring Conventions
- kvalitně pojmenované **identifikátory**
- čtivé a smysluplné **komentáře**
- **PEP 20** - The Zen of Python
- **PEP 440** - Version Identification

21

PEP = Python Enhancement Proposal

- Původně = návrhy na vylepšení jazyka a knihoven
- Fakticky = schválené vylepšení plus návrhy:
  - **Schválené/aktivní/finální**
    - **Standardizační** (*standards track*) ... normativní, jsou součástí jazyka a/nebo standardní knihovny
    - Procesní (*process*) ... řízení procesu schvalování PEP
  - **Informační** historické atd. ... nejsou normativní
  - Dočasně schválené
  - Zastaralé
  - Otevřené/zvažované
  - Odmítnuté/stažené
- **PEP 0** ... seznam všech

22

Další zdroje/inspirace

- **Google Python Style Guide** (github.com, github.io)
- Reitz & Schlusser: Hitchhiker's Guide to Python Style Guide
- Další (obecná) paradigmata
- Další programovací jazyky
  - C++
  - Java
  - ...

23

PEP 8 - Style Guide for Python Code

jako základ pro čistý styl

24

PEP 8 - Style Guide for Python Code

= definice pravidel formátování

- Web peps.python.org
- Čitelný web pep8.org
- Srozumitelné vysvětlení Real Python

- IDE – obvykle silná podpora (dtto výstupy z AI nástrojů)
- Také CLI nástroje (pycodestyle, flake8, ...)

25

PEP 8
základy formátování kódu

26

PEP 8 – znaková sada

- Python pravidla: **UTF-8** (Python 3); ASCII (Python 2)
- Tj. netřeba: `# -*- coding: utf-8 -*-`
- PEP 8 pravidla:
  - pouze anglická abeceda a číslice: **a .. z, A .. Z, 0 .. 9**
  - speciální znaky: **!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~**
  - mezery (\x20) a konce řádku (\n, \x0A, \r, \x0D)
  - nikdy TAB (\t, \x09) a FormFeed (\f, \x0C, Ctrl + L)
  - nikdy Unicode "exotické" mezery (\x2000, \x200A atd.)

27

PEP 8 – znaková sada

- Důsledek: „neanglické" znaky pouze:
  - vlastní jména (lidi, instituce, místa, ...)
  - výstupní text (textové konstanty) pro uživatele
  - textové konstanty pro analýzu dat

!!!

28

PEP 8 - odsazování

- Odsazování ... **4 mezery (a násobky)**
  (chytrý editor/IDE konvertuje z klávesy TAB)
- Odsazování u rozdělených řádek – varianty:
  - Zarovnání pod otevírací závorku
  - Odsazení 4/8 mezery
- Nejrychlejší řešení ... automatické formátování:
  - PyCharm : **Ctrl + Alt + L**
  - Visual Studio Code: **Shift +Alt + F** (alt. Ctrl/K, Ctrl/F)

!!!

29

PEP 8 - odsazování

```
if color == "red":
    stop_vehicle()

total_value = (alfa
               + beta
               + gamma)

def compute_angle (alfa, beta,
                   gamma, delta)
```

Odsazení

Zarovnání pod otevírací závorku

Zarovnání pod otevírací závorku

30

PEP 8 - odsazování

```
First_angle = compute_angle (
      alfa, beta,
      gamma, delta)

First_angle = compute_angle (
      alfa,
      beta,
      gamma,
      delta
)

First_angle = compute_angle (alfa, beta,
                             gamma, delta)
```

Odsazení (PEP8)

Odsazení (doporučení)

Zarovnání pod otevírací závorku

31

PEP 8 - odsazování

Dvojnásobné odsazení zalomeného řádku

```
for count, value in enumerate(
        transform(input_values)):
    print(f"Position {count}: {value}")
```

Standardní odsazení vnořených příkazů

32

PEP 8 – délka řádky

- řádky s kódem <= 79 znaků, Docstring <= 72 znaků
- Alt.: řádky s kódem <= 99 znaků, Docstring <= 72 znaků
- Python Standard Library řádky s kódem < =79 znaků
- Často výjimky (119 nebo 120 znaků, např. PyCharm)
- Pozn. pro nastavení: *vertical edge; hard wrap; rules*

33

PEP 8 – rozdělování řádky

- Rozdělení řádky … předchozí řádka končí znakem `\` (nedoporučuji, dtto Google)
- Rozdělení řádky … **otevřené závorky** `()[]{}` (preferované)
- Pokračovací řádky – varianty pro odsazení:
  - Zarovnání pod otevírací závorku
  - Odsazení 4/8 mezery & otevírací závorka jen s koncem řádky
- Pokračovací řádky … začínají případným binárním operátorem (*Knuth's method*)

34

PEP 8 – rozdělování řádky

```
angles = [alfa,
          beta,
          gamma]

def compute_angle (alfa, beta,
                   gamma, delta):

for count, value in enumerate(
        transform(input_values)):
    print(f"Position {count}: {value}")

for count, value in \
        enumerate(transform(input_values)):
    print(f"Position {count}: {value}")
```

Rozdělení řádky se závorkami

Rozdělení řádky zpětným lomítkem

35

PEP 8 – složené příkazy

- Vždy složené příkazy na více řádků
- Tj. nikdy 2 příkazy na jednu řádku; nikdy `;`
- Pozn. PEP 8 povoluje výjimku pro jednoduché případy; nedoporučuji

ANO:
```
if color == "red":
    stop_vehicle()
```

NE:
```
if color == "red": stop_vehicle()
```


36

PEP 8 – prázdné řádky

Mezi funkcemi/třídami	… 2 řádky	→ 3 řádky
Mezi metodami	… 1 řádka	→ 2/3 řádky
Mezi a vnořenými funkcemi	… 1 řádka	→ 2/3 řádky
Oddělení úseků souboru	… 1 řádka	→ 2 řádky
Oddělení logických úseků kódu	… 1 řádka	
Pro zvýšená čitelnosti kódu	… 1 řádka	

- PEP8 povoluje, ale nedoporučuji: Ctrl/L
- **Doporučené výjimky**

37

PEP 8 – prázdné řádky

- **Doporučené výjimky**
- Nastavení výjimek v konfigurace IDE nástroje (CLI nástroje):
- PEP 8: **E302** expected 2 blank lines, found 3
- PEP 8: **E303** too many blank lines
- PEP 8: **E305** expected 2 blank lines after class or function definition, found 3

38

PEP 8 - Řetězce

- Jednoduché řetězce: uvozovky `"` nebo apostrof `'`
- Doporučuji pro čitelnost: uvozovky `"`
- Víceřádkové řetězce: 3x apostrof `''' text '''`
- Docstring: 3x uvozovky `""" text """`

```
"""\
text
"""
```

!!!

39

PEP 8 - Mezery ve výrazech a příkazech

40

PEP 8 – ANO, používat mezery

- mezera obecně zvyšuje přehlednost a čtivost
- **za** čárkou `,` , ~~středníkem~~, dvojtečkou `:`
- **oboustranně** u binárních operátorů

```
=  +  -  *  /  <  ==  >  <=  >=  !=  :  ::
in  not  in  is  is not  and  or  not  ->
```

- složité výrazy – obvykle jen kolem nejméně prioritních operátorů

41

PEP 8 – NE, nepoužívat mezery

- na konci řádky
- **za** otevírací závorkou: `( [ {`
- **před** zavírací závorkou: `) ] }`
- před čárkou `y, x = x, y`
- vč. čárky před zavírací závorkou `(1,)`
- před středníkem

42

PEP 8 – NE, nepoužívat mezery

- před dvojtečkou u příkazů `if x == 4:`
- před dvojtečkou u type hinting
- před závorkou u volání funkce/metody `fce(x)`
- před závorkou u indexování a řezů `array[1:3]`
- vícenásobné mezery

→ **výjimka "tabulka" (deklarace/volání funkce)**
→ **výjimka "tabulka" (definice strukturovaných konstant)**
→ **výjimka "tabulka" type hinting (deklarace funkce)**

43

PEP 8 – NE, nepoužívat mezery

```
if SERVER == "manual":
    OPCUA_SERVER   = "10.100.0.168"
    OPCUA_PORT     = "4840"
    OPCUA_USER     = "user"
    OPCUA_PASSWORD = "Heslo123+"
    OPCUA_NAMESPACE = "3"
    OPCUA_NAME     = ""
    OPCUA_ROOT     = "DataMES„
elif …
```


44

Výjimky z NE, nepoužívat mezery

- ANO, používat mezery: oboustranně u rovnítka = pro implicitní hodnoty parametrů funkce s (!!!) type hinting
- NE, nepoužívat mezery: oboustranně u rovnítka = pro implicitní hodnoty parametrů funkce bez type hinting

⇨ **Doporučení výjimka … ANO, používat mezery**

- NE, nepoužívat mezery: oboustranně u rovnítka = u pojmenovaných argumentů volání funkce

⇨ **Doporučená výjimka … ANO, používat mezery**

45

Výjimky z NE, nepoužívat mezery

```
def opcua_combine_name(
        root_path: str = "\\",
        name_space: str = "ns=2",
        node_short_name: str = "arm",
        delimiter:  str = ";"
) -> str:
    ""
```

PEP8

```
def opcua_combine_name(
        root_path:       str = "\\",
        name_space:      str = "ns=2",
        node_short_name: str = "arm",
        delimiter:       str = ";"
) -> str:
    ""
```

doporučená výjimka

46

Výjimky z NE, nepoužívat mezery

```
def opcua_combine_name(
        root_path="\\",
        name_space="ns=2",
        node_short_name="arm",
        delimiter=";"
) -> str:
    ""
```

PEP8

```
def opcua_combine_name(
        root_path:       = "\\",
        name_space:      = "ns=2",
        node_short_name: = "arm",
        delimiter:       = ";"
) -> str:
    ""
```

doporučená výjimka

47

PEP 8 – ukončující čárky

- Tj. ukončující čárka před uzavírací závorkou
- Povinně – odlišení závorek u výrazu a inicializaci jednočleného tuple

```
FILES = ("setup.cfg",)
```

- Volitelně – definice seznamů, tuple, slovníků, parametrů, ale musí být víceřádkové

48

Komentáře (nejen podle PEP 8)

49

Kdy použít komentáře

- Dokumentace kódu = docstring (soubory, třídy, metody/funkce)
- Zvýšení čitelnosti kódu
- Nejasný kód; obtížně pochopitelný kód
- Speciální příznaky (TODO atd.)
- Pokyny/příkazy pro IDE, kontrolní nástroje atd.

- **Ideálně**: Docstring + speciální příznaky + pokyny pro IDE
- **Ideálně**: ostatní komentáře jen střídmě, platí „méně je více“

50

Kdy nepoužít komentáře

- Nikdy – “odstranění” kódu, “zakomentování” kódu;
  výjimky: pokusný, vyvíjený nebo a jednorázový kód
  („trvanlivost“ < 1 den)

- Neudržované komentáře nerespektující úpravy kódu (!!!)

- Nekomentujte to, co je přímo v příkazu.

NE:
```
# The total number of elements is count + 1.
count = count + 1
```


51

Jak redukovat komentáře

- **Refaktoring identifikátorů / názvů**:
  Často lze nahradit komentář vhodným názvem proměnné / funkce / metody.

- **Refaktoring struktury kódu:**
  Často lze nahradit komentář voláním vhodně pojmenované (nově vytvořené) funkce / metody realizující celý blok komentovaného kódu.

52

Jak psát komentáře

- **Anglicky**
- Gramaticky správně včetně spellingu, gramatiky a interpunkce.

⇨ Komentář musí být normálně čitelný anglický text.

- AI nástroje:
  - podporují tvorbu komentářů + Docstring
  - nutno kontrolovat
  - často nutno „dočistit“

53

Python komentáře

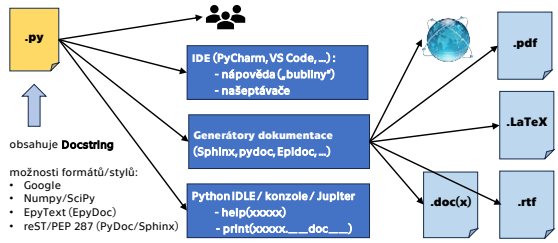
- **Docstring** = technická dokumentace uvnitř kódu

- **Komentáře v kódu** (zvýšení čitelnosti kódu, …)
  - blokové
  - jednořádkové
  - popř. obsahují speciální příznaky (TODO atd.)

- **Speciální komentáře**
  - příkazy (obvykle jednořádkové; přesná syntaxe)

54

Docstring - použití



55

Docstring komentáře

- Docstring = **technická dokumentace uvnitř kódu**:
  - balíčky (packages)
  - moduly / soubory
  - programy / skripty
  - třídy
  - metody / funkce
- Docstring:
  - **jednořádkový**: pro pomocné třídy / metody / funkce
  - **víceřádkový**: pro veřejné (popř. hlavní) částí kódu

56

Docstring komentáře

- PEP 257 – Docstring konvence
  - určuje **obsah**, neurčuje přesný formát
- Docstring podléhají pravidlům odsazování.
- PEP8 Docstring: 3x uvozovky `""" text """` (!!!)

```
"""\
text
"""
```


57

Docstring formát

- Jednořádkový: `"""short-description"""`
- Víceřádkový (obvyklejší):

```
"""\
one-line summary line

… other information
"""
```

Prázdná řádka za první řádkou

Prázdná řádka na konci

58

Metody a funkce - Docstring

- **stručný popis**
- **úplný popis**
- **parametry**
  - jméno
  - typ
  - popis
  - jsou-li nepovinné [optional]
  - implicitní hodnoty
  - možnost použití klíčového slova [keyword])
- **návratová hodnota** (typ, popis)
- **výjimky** (vznikající v metodě/funkci)

59

Třída - Docstring podle PEP 257

- **stručný popis**
- **úplný popis** (u podtříd uvést nadtřídu a popsat rozdíly)
- **veřejné atributy (jméno, typ, popis)**
- **veřejné metody třídy** (u podtříd uvádět pro zděděné metody, popsat rozdíly a uvést jsou-li override nebo extend)
- Pozn. konstruktor třídy se dokumentuje u metody __init__()

60

Modul/soubor - Docstring (PEP 257)

- **stručný popis**
- **úplný popis**
- **veřejné třídy** (jednořádkový popis, detailně až u implementace)
- **veřejné výjimky** (jedna řádka popis, detailně až u implementace)
- **veřejné funkce** (jedna řádka popis, detailně až u implementace)
- **veřejné konstanty a globální proměnné (jméno, typ, popis)**

61

Balíček [package] – Docstring (PEP 257)

- **stručný popis**
- **úplný popis**
- **veřejné vnořené moduly**
- **veřejné vnořené balíčky** [subpackage]
- **veřejné třídy** (jednořádkový popis, detailně až u implementace)
- **veřejné výjimky** (jednořádkový popis, detailně až u implementace)
- **veřejné funkce** (jednořádkový popis, detailně až u implementace)
- **veřejné konstanty a globální proměnné (jméno, typ, popis)**
- uvádí se v modulu __init__.py celého balíčku

62

Program/skript – Docstring (PEP 257)

- Cca shodný s textem, který program vytiskne jako svůj návod k použití [help]; může být dlouhý i několik obrazovek
- Podrobný natolik, aby i nový uživatel použil program správně
- Kompletní popis všech parametrů [quick reference]
- **stručný popis**
- **úplný popis**
- **syntaxe a parametry příkazové řádky**
- **používané proměnné prostředí operačního systému**
- **používané konfigurační soubory**

63

Docstring - další (nepovinné) údaje

- vedlejší efekty (side-effect)
- omezení při použití
- popis implementace
- použité zdroje, informace atd.

64

Docstring komentáře - formáty

- Google
- NumPy / SciPy
- EpyText (EpyDoc)
- reStructuredText (reST/PEP 287/Sphinx)

65

DocString formáty

- **Google**: několik sekcí
  jméno sekce **:** \n – odsazení – jméno (typ) **:** – popis
- **NumPy / SciPy**: několik sekcí:
  jméno sekce – \n podtržení pomlčkami \n
  jméno parametru atd. – typ – odsazení – popis
- **reStructuredText(reSt/PEP 287/Sphinx)**: několik řádků
  : klíčové slovo – mezera – jméno **:** popis nebo typ
- **Epytext (EpyDoc)**: několik řádků, podobné JavaDoc
  @klíčové slovo/pole – mezera – jméno **:** popis nebo typ

Bez možnosti formátování

Markup jazyky = možnost bohatého formátování

66

Google Docstring příklad

```
def calculate_area(radius):
    """
    Calculates the area of a circle.

    Args:
        radius (float): The radius of the circle.

    Returns:
        float: The area of the circle.

    Raises:
        ValueError: If the radius is negative.

    Examples:
        >>> calculate_area(5)
        78.53981633974483
    """

    if radius < 0:
        raise ValueError("Radius cannot be negative.")
    return 3.14159 * radius ** 2
```


67

Google sekce

- Stručný a detailní popis modulu / třídy / metody / funkce

Args: jméno argumentu, typ, popis
Returns: typ a popis návratové hodnoty
Yields: typ a popis návratové hodnoty
Raises: seznam výjimek a důvod vzniku

Classes: jména a krátký popis třídy
Methods: jména a krátký popis metod třídy
Functions: jména a krátký popis metod funkce
Attributes: jméno atributu, typ, popis

68

Google sekce

Examples: (nepovinné) ukázky použití (volání)

See Also: (nepovinné) související třídy, typy, funkce atd.

Note: (nepovinné) další poznámky

References: (nepovinné) odkazy na externí zdroje

69

Numpy Docstring příklad

```
def calculate_area(radius):
    """
    Calculate the area of a circle.

    Parameters
    ----------
    radius : float
        The radius of the circle.

    Returns
    -------
    float
        The area of the circle.

    Raises
    ------
    ValueError
        If the radius is negative.

    Examples
    --------
    >>> calculate_area(5)
    78.53981633974483
    """

    if radius < 0:
        raise ValueError("Radius cannot be negative.")
    return np.pi * radius ** 2
```


70

NumPy/SciPy sekce

Parameters jméno argumentu, typ, popis
Returns typ a popis návratové hodnoty
Yields typ a popis návratové hodnoty
Raises seznam výjimek a důvod vzniku
Attributes jméno atributu, typ, popis
Methods jména a krátký popis metod třídy

71

NumPy/SciPy sekce

See Also (nepovinné) související třídy, typy, funkce ...
Notes (nepovinné) další poznámky
Examples (nepovinné) ukázky použití (volání)
Warnings (nepovinné) upozornění spojená s použitím

- Řada dalších direktiv (struktura textu, formátování textu)
- Podpora LaTeX-u

72

reST Docstring příklad

```
def calculate_area(radius):
    """
    Calculate the area of a circle.

    :param radius: The radius of the circle.
    :type radius: float

    :return: The area of the circle.
    :rtype: float

    :raises ValueError: If the radius is negative.

    :Example:

    >>> calculate_area(5)
    78.53981633974483
    """

    if radius < 0:
        raise ValueError("Radius cannot be negative.")
    return 3.14159 * radius ** 2
```


73

reST/PEP 287/Sphinx klíčová slova

:param jméno a popis argumentu
:type typ parametru
:return popis návratové hodnoty
:rtype typ návratové hodnoty
:raises seznam výjimek a důvod vzniku
:func popis funkce
:method popis metody
:ivar jméno a popis proměnné / atributu
:ivartype typ proměnné / atributu

74

reST/PEP 287/Sphinx klíčová slova

:example (nepovinné) ukázky použití (volání)
:note (nepovinné) další poznámky
:warning (nepovinné) upozornění spojená s použitím
:todo (nepovinné) seznam dalších prací
:seealso (nepovinné) související třídy, typy, funkce ...

- ... a další
- Bohaté možnosti formátování, podobné Markdown
- Řada dalších direktiv (struktura textu, formátování textu)

75

EpyText Docstring příklad

```
def calculate_area(radius):
    """
    Calculate the area of a circle.

    @param radius: The radius of the circle.
    @type radius: float

    @return: The area of the circle.
    @rtype: float

    @raise ValueError: If the radius is negative.

    @example:
    >>> calculate_area(5)
    78.53981633974483
    """

    if radius < 0:
        raise ValueError("Radius cannot be negative.")
    return 3.14159 * radius ** 2
```


76

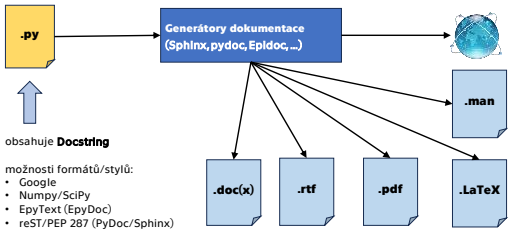
Epytext klíčová slova [field / tag]

@param	@requires	@summary	@author
@type	@precondition	@todo	@contact
@return	@postcondition	@note	@organisation
@rtype	@invariant	@attention	@copyright
@keyword	@deprecated	@bug	@license
@raise	@permission	@see	@version
		@since	@status

- Bohaté možnosti formátování, podobné, ale slabší, než Markdown nebo reST

77

Docstring - použití



obsahuje **Docstring**

možnosti formátů/stylů:
- Google
- Numpy/SciPy
- EpyText (EpyDoc)
- reST/PEP 287 (PyDoc/Sphinx)

78

Generátory dokumentace

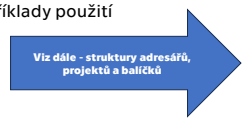
- Sphinx https://www.sphinx-doc.org/
- Epydoc https://epydoc.sourceforge.net/
  už není udržován (od 2013)
- Doxygen https://www.doxygen.nl/index.html
  + doxypypy https://github.com/Feneric/doxypypy
- MkDocs https://www.mkdocs.org/
  https://pypi.org/project/mkdocs/
- pdoc https://github.com/mitmproxy/pdoc
- pdoc3 https://pdoc3.github.io/pdoc/
- pydoctor https://github.com/twisted/pydoctor

79

Dokumentace projektů/balíčků

~ dokumentace vně kódu

- Adresářová struktura více souborů; existuje více variant
- Obsahuje:
  - dokumentaci
  - instalační návody
  - popř. testy, licence, návody, příklady použití
- Realizace:
  - adresář
  - GIT repozitář
  - distribuční balíček [package]



80

Python komentáře

- **Docstring** = technická dokumentace uvnitř kódu
- **Komentáře v kódu** (zvýšení čitelnosti kódu, ...)
  - blokové
  - jednořádkové
  - popř. obsahují speciální příznaky (TODO atd.)
- **Speciální komentáře**
  - příkazy (obvykle jednořádkové; přesná syntaxe)

81

PEP 8 - komentáře

- **Blokové**
  - `#` + **mezera** + komentář
  - na samostatném řádku (!!!)
  - předchozí řádek před příslušným příkazem (!!!)
  - obvykle jeden prázdný řádek před komentářem
- **Jednořádkové** (inline)
  - nejméně **2x mezera** + `#` + **mezera** + komentář
  - na konci řádku

82

Blokové a inline komentáře

```
# We use a weighted dictionary search to find
# out where i is in the array.  We extrapolate
# position based on the largest number
# in the array and the array size and then
# do binary search to get the exact number.
```

2x mezera

```
if i & (i-1) == 0:  # True if it's 0 or power of 2.
```


83

Python komentáře

- **Docstring** = technická dokumentace uvnitř kódu
- **Komentáře v kódu** (zvýšení čitelnosti kódu, ...)
  - blokové
  - jednořádkové
  - popř. obsahují speciální příznaky (TODO atd.)
- **Speciální komentáře**
  - příkazy (obvykle jednořádkové; přesná syntaxe)

84

Speciální komentáře

A. **speciální komentáře** [někdy též codetags]

- Pozn. existuje odmítnuté doporučení PEP 350

B. **řídící příkazy** pro systémy statické analýzy, IDE atd.

85

„Řídící" komentáře pro IDE atd.

#NOSONAR

#NOINSPECTION xxx ... PyCharm
https://www.jetbrains.com/help/pycharm/disabling-and-enabling-inspections.html#comments-ref

 # pylint: disable=xxx
https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html

 # pyright: ignore [xxx] Pylance/Pyright
https://github.com/microsoft/pyright/blob/main/docs/comments.md
https://github.com/microsoft/pyright/blob/main/docs/configuration.md

 # type: ignore [xxx] ... MyPy / PEP484
https://mypy.readthedocs.io/en/stable/error_code_list.html
https://mypy.readthedocs.io/en/stable/error_code_list2.html#error-codes-optional

86

Speciální příznaky [codetags]

# TODO	... popis toho, co je nutné dodělat
# FIXME	... popis **interní** chyby a co je nutné opravit
# REALLY	... zdůraznění, že **neobvyklý** kód je určitě správně
# MAGIC	... triky, obtížné na pochopení, atd.
# HACK	... popis speciálních postupů
# WORKAROUND	... popis **externí** chyby a jejího napravení
# REQUIRES	... popis, co je vyžadováno nad rámec kódu
# ASSUMPTION	... popis, co kód předpokládá, obvykle o proměnných
# SIDE-EFFECT	... používaný nebo využívaný vedlejší efekt
# DIRTY	... opravdu nevhodný kód, který je ale nezbytný

87

Komentář `#TODO`

- popis toho, co je nutné dodělat (doprogramovat)

```
color = WHITE   # TODO: add more functionality
```

alternativně:

```
color = WHITE   # TODO: add color selection
```

- často podpora v IDE
- možné kombinovat s odkazem na systém sledování chyb (Atlassian Jira, BugZilla, ...), tedy ID příslušného záznamu
- možné kombinovat s iniciálami autora, datumem atd.

88

Komentář `#TODO`

- možné kombinovat s výjimkou `NotImplementedError`

```
def set(self, channel):  # TODO: add implementation
    raise NotImplementedError("Can't use 'set'.")
```

- Správné použití této výjimky je jen v abstraktní metodě (hlavní) třídy, lze ale použít i takto explicitně.
- Vždy uvádět smysluplný parametr (tj. text)!

89

Komentáře `#FIXME` a `#REALLY`

 # FIXME: blabla ... popis **interní** chyby a popř. toho, co a jak je nutné opravit; někdo používá alternativně jako popis nutného zlepšení (nikoliv chyby)

 # REALLY: blabla .... krátce, že kód je vlastně správně (ačkoliv kód vypadá, že je chybný, resp. podezřelý)

```
except Exception:   # REALLY: catch all exceptions

pass   # REALLY: do nothing
```


90

Komentáře – speciální příznaky

 # MAGIC: blabla .... triky, obtížné na pochopení, atd.

 # HACK (for ... browser/OS/library ...): blabla... popis speciálních postupů ... (nebo nic, je-li to jasné z kódu)

 # WORKAROUND (for ... what ...): ... popis chyby a jejího napravení ... (nebo nic, je-li to jasné z kódu)

91

Komentáře – speciální příznaky

REQUIRES: blabla … popis, co je vyžadováno (další moduly, nastavení OS, data, další programy atd.)

```
# REQUIRES installation of module poppler
import pdfplumber
```


92

Komentáře #ASSUMPTION

 # ASSUMPTION: blabla … popis, co kód předpokládá, obvykle o hodnotách proměnných, parametrů, mezivýsledků atd.

```
#ASSUMPTION: velocity is possitive
```

- alternativně – zahrnout do komentáře celé funkce / metody / třídy / modulu).

93

Komentáře #ASSUMPTION

- lépe – použít příkaz `assert`

```
assert velocity > 0, f„bad value of velocity (={velocity})"
```

- **Předpoklad** [příkaz `assert` nebo #ASSUMPTION]
  - vyžadovaný logikou výpočtu
  - by měl být vždy splněn, tedy vyloučení nemožných situací.
- **Výjimka** [exception]
  - chybový stav, který může někdy nastat.
  - tj. příkazy a objekt: [`raise` / `try` / `Exception`]

94

Komentáře #SIDE-EFFECT a #DIRTY

- Nejlepší je psát kód bez vedlejších efektů a jiného nevhodného řešení. Nelze-li jinak, nutno precizně okomentovat.

 # SIDE-EFFECT: blabla …. popsat používaný nebo využívaný vedlejší efekt deklarované nebo používané funkce

 # DIRTY: blabla …. Zdůraznit a okomentovat opravdu nevhodný kód, který je ale nezbytný

95

Jmenné konvence, identifikátory atd. (nejen PEP 8)

96

PEP 8 – jmenné konvence

- Popisné (!!!) identifikátory
- Povinně ASCII, resp. **písmena, číslice, pomlčka**
- **Celá slova**; jen výjimečně zkratky a akronymy
- Preferovaně **anglicky**
  - včetně spellingu
  - obvykle bez gramatické preciznosti (členy, "to", "of", "'s")
- **Popis podle funkcionality**

97

Nevhodné pojmenování

- Nepopisovat podle implementace
- Nepopisovat podle návratového typu (tzv. maďarská notace)
- Nikdy nepoužívat matoucí/zaměnitelná slova znaky:
  - klíčová slova
  - velké písmeno i I (tedy „íííí“)
  - malé písmeno l l (tedy „el“)
  - velké písmeno o O (tedy „óóó“)

98

Nevhodná jména

- Jednopísmenné názvy, zkratky, akronymy
- Nevyslovitelná a nesmyslná slova včetně zkratek
- Neslušná slova (pozn. kulturně závislé)
- Slangová slova
- Názvy obvyklých knihoven a jejich častých funkcí
- Matoucí název neodpovídá významu

99

PEP 8 - Jmenné konvence - styly

Styl jména	Entity jazyka Python	
lowercase lowercase_name	proměnné funkce moduly / soubory	atributy (tříd) metody (tříd)
CAPITAL CAPITAL_LETTERS	konstanty	
Capitalised CapitalisedWords	třídy	typy
CapitalisedWordsError	výjimky	
lowercasename	balíčky / adresáře	

100

PEP 8 – jmenné konvence

- **Speciální globální proměnné** *(„dunder“)* `__xyz__`
- **Speciální metody tříd** *(„dunder“)* `__xyz__`
- **Privátní metody a členy tříd a modulů** `_alfa`
- **Metody instancí – první argument** `self`
- **Metody tříd – první argument** `cls`
- Výjimky z jmenných konvencí – starý kód nebo kompatibilita s implementací v jiném jazyce (PIL, logger, ...)

101

Proměnné, atributy, třídy

- **Podstatné jméno**, případná příčestí následují; bez členů
- **Číselné hodnoty** ⇨ nejlépe počitatelná podstatná jména nebo jasná „číselná“ sémantika:
  `age` `year` `item_count` `order_number` `arm_length` `wheel_diameter` ...
- **Řetězcové hodnoty** nejlépe:
  `xxx_name` `xxx_title` `message_xxx_name` ...

102

Funkce a metody - konvence

- **Sloveso**, nejlépe akční sloveso:
  `load_xxx` `fetch_xxx` `run_xxx` `compute_xxx` `move_xxx` ...
- **Predikáty**, nejlépe sloveso navozující odpověď ANO/NE:
  `is_xxx` `has_xxx` `was_xxx` `can_xxx` ...
- **Atributy metod**, slovesa get, set, has:
  `get_xxx` `set_xxx` `has_xxx` ...
- Konzistentí použití všech slov (!!!)

103

Matematika, fyzika, …

- Obvykle pracují s jednopísmennými proměnnými
- Dále si "pomáhají" dalšími abecedami, indexy, …
- Nahradit plnohodnými identifikátory
- Fyzikální jednotky … vše v jednotkách podle SI, jinak jednotku uvést v názvu, obvykle na konci identifikátoru

104

Jmenné konvence - výjimky

- Zkratky a akronymy, které jsou opravdu široce známé, popř. normalizované (ISO/EN/ČSN):
  - měny, státy, ekonomika: `czk` `eur` `vat` `kpi` `id` …
  - IT (`tcp` `udp` `url` `http` …)
  - aplikační doména
- Jednopísmenné/zkrácené v opravdu krátkém (!!!) kódu :
  - pro indexaci bez dalšího významu (`i` `j` `k` `x` `y` …)
  - pomocné proměnné (`tmp` `txt` `len` …)

105

Příkaz import, použití modulů a balíčků

106

Moduly a balíčky

- **Modul** v Pythonu:
  - Rozdělení problému na menší části
  - Znovupoužitelný kód
  - Využití kódu jiných autorů
- **Balíček** (*package*) v Pythonu:
  - více modulů (!!!), popř. více „pod-balíčků"
  - tečková notace
  - adresářový strom
  - povinně soubor `__init__.py`
  - použití: distribuce kódu, knihovny, API, …

107

Moduly a balíčky

- **Technické varianty modulu** v Pythonu:
  1. Python soubor
  2. C modul
  3. Vestavěné (built-in) moduly
- Příkaz `import` – zpřístupnění obsahu modulů + balíčků:
  - Moduly – hledá soubory
  - Balíčky – hledá adresáře se souborem `__init__.py`
    – pak hledá soubory a podadresáře

108

Příkazy `import` – pořadí hledání

- moduly již v paměti (proměnná `sys.modules`)
- Aktuální adresář
- PYTHONPATH proměnná operačního systému
- Pevná místa instalace Python (podle operačního systému)
- Pozn: Anaconda a VENV (Virtual Environments) mění tyto cesty

109

Pevná místa instalace Python

- Windows
  - c:\Program Files\PythonXY\Lib\
  - c:\Users\USERNAME\AppData\Roaming\Python\PythonXY\site-packages
- Linux
  - /usr/local/lib/pythonX.Y/site-packages
  - /usr/local/lib/pythonX.Y/dist-packages
  - /usr/lib/pythonX.Y
  - /usr/lib/pythonX.Y/lib-dynload
  - /home/user/.local/lib/pythonX.Y/site-packages
  - /usr/lib/python3/dist-packages

110

PEP 8 - Příkazy `import`

- Vždy na začátku souboru za docstring a komentáře, tj. před konstanty a globální proměnné
- Absolutní import – preferovaný protože čitelnější

```
from some_module import some_class
```

- Relativní import – akceptovatelné, ale nedoporučuji

```
from .some_module import some_class
from ..some_package import some_function
from . import some_class
```


111

PEP 8 – Pořadí příkazů `import`

- Každý modul/balíček na samostatné řádce (doporučuji stejně, je-li import více funkcí, objektů, atd.)
- Příkazy import rozděleny na sekce podle druhů modulů/balíčků
- Sekce oddělené prázdnou řádkou
- Popř. každá sekce abecedně řazená
- Speciální sekce – příkaz future: `from __future__ import ....`

112

PEP 8 – Pořadí příkazů `import`

PEP 236

0 { • příkaz future → 0

PEP8

1 { • standardní knihovna → 1

2 { • dobře známé knihovny třetích stran → 2
• ostatní knihovny třetích stran → 3

3 { • vlastní obecné knihovny → 4
• moduly projektu → 5

Doporučení

- Popř. v rámci sekcí abecedně

113

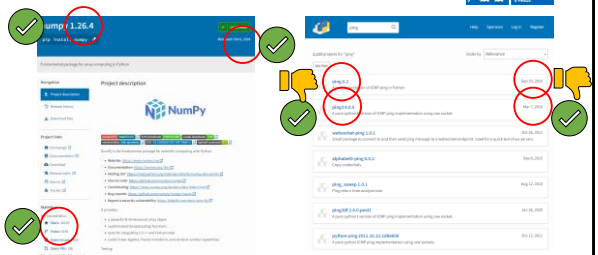
Druhy modulů a balíčků

- standardní knihovna

  https://docs.python.org/3/library/index.html
- knihovny třetích stran
  - příkaz `pip install xxxxx`
  - příkaz `python -m pip install xxxxx`
  - příkaz `conda install xxxxx`
  - PyPi = Python Package Index ... *https://pypi.org/*
  - Anaconda repository ... https://anaconda.org/anaconda/repo

114

PyPi –„dospělé“ moduly



115

PEP 8 - Příklady `import`

- **Nikdy import všech pomocí hvězdičky**
  protože se zakrývá „původ" importovaných objektů

NE: `from xxx import *`

- Výjimka: moduly projektu

ANO: `from parking_turtle_constants import *`

116

Příkazy `import` (Google)

- Importovat **pouze pro balíčky a moduly**
- Neimportovat individuální třídy, funkce, konstanty
  protože v delším kódu zakrývá „původ" používaných tříd, ...
- Pozn.: velmi často nedodržované

ANO:
```
import pathlib
import matplotlib.pyplot
```

NE:
```
import pathlib.Path
from pathlib import Path
from matplotlib import pyplot
from matplotlib.pyplot import show
```


117

Import pouze modulů/tříd

- Důsledek:

```
import myextraclass
import foo.bar.myclass
```

Explicit is better than implicit.
(Zen of Python)

A pak použití celých jmen:

```
data_object = myextraclass.MySolver()
data_object = foo.bar.myclass.FooBarSolver()
```


118

Import „čehokoliv"

- Důsledek:

```
from myextraclass import MySolver
from foo.bar.myclass import FooBarSolver
```

... 200 řádků kódu ... a pak:

Kde se vlastně vzaly třídy `MySolver()` a `FooBarSolver`

```
data_object = MySolver()
data_object = FooBarSolver()
```


119

Příkazy `import` (Google)

- Výjimky pro **import individuálních tříd**, funkcí, konstant:
  - `typing`
  - `collections.abc`
  - `typing_extensions`
- Příklady:

```
from typing import List
from collections.abc import Iterable
```


120

Příkazy `import` (Google)

- Použití plných jmen pro moduly z balíčků
- Pozn. zastaralé rozlišování importu z balíčků a „pod-balíčků"
- Tj. **preferovat tečkovou notaci** před `from ... import`

ANO:
```
import matplotlib.pyplot
import package2.subpackage1.module5
```

NE:
```
from matplotlib import pyplot
from package2.subpackage1 import module5
```


121

Příkazy `import .. as` (Google)

- **Nepřejmenovávat při importu** pomocí `as`
- Výjimky:
  - Obvyklé zkratky pro dobře známé knihovny třetích stran
  - Opravdu dlouhá jména
  - Řešení nekompatibilit různých verzí

ANO:
```
import matplotlib.pyplot as plt
import numpy as np
```

NE:
```
import utils.terminal_width as terminal_width
```


122

Příkazy `import .. as` (Google)

- **Povolené výjimky** pro přejmenování importu:
  - datetime dt
  - matplotlib.pyplot plt
  - multiprocessing mp
  - numpy np
  - pandas pd
  - seaborn sns
  - tensorflow tf
  - tkinter tk

123

Souvislosti příkazu `import`

- PEP 20 - The Zen of Python

„Explicit is better than implicit."

⇦ Nepoužívat import pomocí hvězdičky
⇦ Nepoužívat import jednotlivých tříd, funkcí, ...
⇦ Použití plných jmen pro importy modulů z balíčků
⇦ Nepřejmenovávat při importu pomocí `as`

124

Struktura souboru s kódem (nejen PEP)

125

Python struktura souboru

Shebang	`#!/usr/bin/env python3`
Kódování (nepovinné)	`# -*- coding: utf-8 -*-`
Docstring – dokumentace	`""" … docstring modulu … """`
Komentář k souboru (nepovinné)	`''' … další popis … '''`
Hlavička souboru (autor atd.)	`__author__ = …`
Import modulů a balíčků	`import …`
Konstanty a globální proměnné	`DEBUG = …`
Třídy a funkce	`def … / class …`
Spuštění programu (u balíčků nepovinné)	`if __name__ = "__main__": …`

126

Python shebang

- Povinné
- První řádek souboru
- Pro Linux/Unix/POSIX určuje interpreter

ANO: `#!/usr/bin/env python3`

NE: `#!/usr/bin/python3`
NE: `#!/usr/bin/env python`
NE: `#!/usr/bin/python`

127

Python kódování souboru

- Volitelné
- Druhý řádek souboru
- Obecně: **UTF-8** (Python 3); ASCII (Python 2)
- Tj. netřeba: `# -*- coding: utf-8 -*-`

128

Docstring modulu (souboru)

- Povinné
- Dokumentace vnořená v programu
- PEP 257 - Docstring Conventions
- Tento text bude zobrazován jako nápověda v IDE atd.
- Docstring: 3x uvozovky `""" text """` (!!!)

```
"""\
text
"""
```

- Pro uživatele kódu (modulu, balíčku, knihovny, …)

129

Komentář na úrovni modulu

- Volitelné
- Nebude součástí nápovědy zobrazované v IDE
- Varianty:
  - Blokový komentář : `#` a mezera
  - Víceřádkový řetězec: 3x apostrof `''' text '''` (!!!)

```
'''\
text
'''
```

- Pro editory/údržbáře kódu (modulu, balíčku, knihovny, …)

130

Python struktura souboru

Shebang	`#!/usr/bin/env python3`
Kódování (nepovinné)	`# -*- coding: utf-8 -*-`
Docstring – dokumentace	`""" … dosstring modulu … """`
Komentář k souboru (nepovinné)	`''' … další popis … '''`
Hlavička souboru (autor atd.)	`__author__ = …`
Import modulů a balíčků	`import …`
Konstanty a globální proměnné	`DEBUG = …`
Třídy a funkce	`def … / class …`
Spuštění programu (u balíčků nepovinné)	`if __name__= "__main__": …`

131

Python hlavička souboru

```
__author__      = "David Koníček"
__maintainer__  = "David Koníček"
__email__       = "david.konicek@cvut.cz"
__copyright__   = "\xa9 2022 CIIRC CTU, Prague."\
                  " All rights reserved."
__license__     = "MIT"
__version__     = "0.1.0"
__date__        = "2023/01/02"
__status__      = "Development"
__credits__     = []
__all__         = []
```

textové řetězce

seznam řetězců

132

Python hlavička souboru

- Metadata programu/modulu
- „Nepsaný standard" pro všechnu programátorskou práci, nejen Python
- Každý programovací jazyk má svoji syntaxi
- Python vychází z pravidel pro Docstring dle Epydoc
  https://epydoc.sourceforge.net/manual-fields.html#module-metadata-variables
- Python vychází z pravidel tvorbu balíčků PyPA (Python Packaging Association)
  https://www.pypa.io/en/latest/
- Rob Knight: Python Coding Guideliness (for Cogent project)
  https://web.archive.org/web/20111010053227/http://jaynes.colorado.edu/PythonGuidelines.html#module_formatting

133

Python hlavička souboru

`__author__` – řetězec (!!!), obsahuje jedno či více jmen (oddělovat čárkami nebo středníkem)

`__maintainer__` – řetězec (!!!), obsahuje jedno či více jmen (oddělovat čárkami nebo středníkem)

`__email__` – řetězec (!!!), obsahuje jedno či více emailových adres (oddělovat čárkami nebo středníkem)

134

Autorská práva a licence

- **Copyright = právo autora**, vzniká automaticky autorovi
- **Licence = právní ujednání**, definuje:
  1. práva užití a obvykle
  2. obvykle omezuje odpovědnost za škodu

- *What's the difference between Copyright and Licensing?*
  https://opensource.stackexchange.com/questions/297/whats-the-difference-between-copyright-and-licensing

135

Autorská práva

- **Copyright = právo autora**; vzniká automaticky ze zákona
- Autorské dílo = nehmotný duševní výtvor, které musí mít znaky autorského díla (tvůrčí činnost; jedinečnost; zhmotněné)
- Legislativa: zákon 121/2000 Sb., popř. předpisy EU, popř. předpisy země užití, popř. celosvětové dohody
- Pozn.: slovo **copyright** a značka **©** jsou zvykově používány (do 1988 v USA závazně), ale v ČR/EU nemají právní význam
- Pro laiky je přijatelný zdroj informací česká Wikipedie
  https://cs.wikipedia.org/wiki/Autorský_zákon_(Česko,_2000)

136

Hlavička souboru: `__copyright__`

`__copyright__`

- **Copyright = právo autora**; vzniká automaticky ze zákona
  ⇨ tento text je deklaratorní, resp. informativní o držiteli práv

- Obvyklý text: držitel práv, rok/roky a deklarace

```
__copyright__ = "\xa9 2022 CIIRC CTU, Prague."\
                " All rights reserved."
```


137

Hlavička souboru: `__copyright__`

`__copyright__` – varianty možného textu:

1) Student: dle ustanovení §60 zákona o školních dílech
   ⇨ rok/roky, fyzický autor, jméno školy, popř. fakulty
2) Zaměstnanec: dle ustanovení §58 zákona o zaměstnaneckých dílech ⇨ rok/roky, zaměstnavatel
3) Speciálně uzavřené smlouvy – podle jejich textu
4) Ostatní případy ⇨ rok/roky, fyzický autor

(*Pozn. nebylo konzultováno s právním oddělením*)

138

Licence = právní ujednání

- Pro laiky je přijatelný zdroj informací a zejména **porovnání** anglická Wikipedie:
  https://en.wikipedia.org/wiki/Software_license
- **OSI Approved Licenses**: obsáhlý výčet licencí volně dostupného software (GNU, MIT, Apache, ...):
  https://opensource.org/licenses/
- Proprietární software přináší mnoho dalších typů licencí
- Praxe přinesla jiné texty pro software (GNU/GPL, MIT, APL, ...) a pro ostatní autorská díla (CC = Cretive Commons, ...)

139

Licence = právní ujednání

- Doporučuji se řídit dle **OSI Approved Licenses** (mezinárodní)
  https://opensource.org/licenses/?categories=international
- Protože jsme členský stát EU (všech 23 jazyků, 27 zemí):
  https://joinup.ec.europa.eu/collection/eupl/eupl-text-eupl-12
- Doporučuji vybírat z licencí: **MIT, GPL, EUPL**
- Další možnost licence - WTFPL, ale pozor, neřeší zřeknutí se odpovědnosti za škody:
  http://www.wtfpl.net/
  https://en.wikipedia.org/wiki/WTFPL

140

Hlavička souboru: `__license__`

`__license__` – výběr textu:

- Základ: váš právní status (student, zaměstnanec, zcela volný)
- Důrazně doporučuji neměnit text a držet se toho, co právníci vymysleli; text je vesměs univerzální ve smyslu různých národních úprav.
- Obvykle řešení:
  - Výběr jedné z licencí: **MIT / GPL /EUPL**
  - standardní zkratka podle webu OSI Approved Licenses
  - přiložený textový soubor s názvem **LICENSE** nebo **LICENSE.TXT** s doslovným (!!!) zněním licence

141

Python hlavička souboru: `__date__`

```
__date__ = "2024-04-03"
```

- volitelné
- datum poslední změny
- textový řetězec (!!!)
- formát podle ISO 8601, resp. ČSN ISO 8601
  ⇨ YYYY-MM-DD

- ideálně automaticky doplňuje verzovací systém (GIT atd.)

142

Hlavička souboru: `__status__`

- Status by měl odpovídat logice číslu verze (alfa, beta, release candidate atd.)

`__status__` – řetězec (!!!), obvykle jedno z

```
"Prototype"
"Development"
"Production"
```


143

Python číslování verzí

`__version__ = "..."` – řetězec (!!!)

- PEP 440 - Version Identification
  https://peps.python.org/pep-0440/
- Inspirováno – sémantické verzování
  https://semver.org/
- Alternativa – kalendářové verzování
  https://calver.org/
- Reálný svět – marketing rozhoduje ⇨ nejrůznější kombinace a texty

144

Obvyklé sémantické verzování

- N.N.N .... číslo.číslo.číslo ... např. 1.43.15 ... jako řetězec
- [major].[minor].[release].[build]

Verze	Význam a použití
MAJOR	změně architektury nebo změna narušující zpětnou kompatibilitu
MINOR	přidání funkcionality při zachováním zpětné kompatibility
RELEASE / PATCH / MICRO	opravy chyb při zachováním zpětné kompatibility a žádné změny funkcionality
BUILD	neustále rostoucí číslo reprezentující vnitřní verzi (málo používané)

145

Číslování verzí podle PEP 440

v	Epoch	!	Release	-	Pre-release	-	Pre-release	-	Dev	+	Local
o	Optional		Required		Optional		Optional		Optional		Optional

`[N!]N(.N)*[{a|b|rc}N][.postN][.devN][+.*]`

- `N!` Epocha (nepovinné) = změna logiky číslování
- `N(.N)*` Vydání [release] několik čísel, obvykle N.N.N
- `a|b|rcN` Předběžné verze [pre-release]
- `.postN` Dodatečná verze [post-release] (nepovinné)
- `.devN` Vývojové verze [development]

- PEP 440 ⇨ nová verze ⇨ PyPa Version specifiers

https://packaging.python.org/en/latest/specifications/version-specifiers

146

Číslování verzí podle PEP 440

`[N!]N(.N)*[{a|b|rc}N][.postN][.devN][+.*]`

release | development / internal / revision

- `.devN` Vývojové verze, lze kombinovat s `a|b|rc|post|dev`
- `aN` Alfa verze = neobsahuje všechnu funkcionalitu
- `bN` Beta verze = veškerá funkcionalita, netestováno
- `rcN` Release Candidate = interně testováno, „skoro" dobré, probíhá externí testování
- finální (produkční) verze; tj. bez `a|b|rc|post|dev`

- Pozn.: *alfa - beta- RC - produkční* verze odpovídá standardním fázím vývojového cyklu software

147

Python hlavička souboru

`__credits__` – seznam řetězců (!!!), **ne**obsahuje autory importovaných modulů, ale obsahuje autory m.j.:

- doporučení a rad
- kteří nahlásili chyby
- použitých fragmentů kódu
- inspirativních textů

```
__credits__ = ["Rob Knight", "Peter Maxwell"]
```


148

Python hlavička souboru

`__all__` seznam (!!!) řetězců jmen tříd, funkcí, konstant atd., které modul **exportuje**

- Tj. jediná část metadat **používaná jazykem Python**
- Zároveň zvyšuje čtivost kódu

149

Příkazy `import` a seznam `__all__`

`from xxx import *` a importovaný modul neobsahuje `__all__` ⇨ importuje opravdu všechno

`from xxx import *` a importovaný modul obsahuje `__all__` ⇨ importuje jen vyjmenované v `__all__`

`import xxx` a aktuální modul obsahuje použití objektů z importovaného modulu ⇨ kontroluje, že použité je v `__all__`

150

Python struktura souboru

Shebang	`#!/usr/bin/env python3`
Kódování (nepovinné)	`# -*- coding: utf-8 -*-`
Docstring – dokumentace	`""" … dosstring modulu … """`
Komentář k souboru (nepovinné)	`''' … další popis … '''`
Hlavička souboru (autor atd.)	`__author__ = …`
Import modulů a balíčků	`import …`
Konstanty a globální proměnné	`DEBUG = …`
Třídy a funkce	`def … / class …`
Spuštění programu (u balíčků nepovinné)	`if __name__ = "__main__": …`

151

Import, konstanty, proměnné, třídy, funkce

- **Importy** – členění do sekcí podle PEP8
- **Konstanty** – názvy podle PEP8; popisné
- (Popř.) **globální proměnné** – názvy podle PEP8; popisné; preferovaně s podtržítkem na začátku
- **Třídy**
- **Funkce**

⇨ V celém modulu není žádný přímo vykonávaný kód

Python program v „klasickém" pojetí

152

"Přímý kód" v souboru

- V celém modulu není žádný přímo vykonávaný kód
- Výjimky:
  - Příkazy import
  - Definice konstant
  - Inicializace globálních proměnných
  - Patička s případným voláním main()
- Další výjimky:
  - Testovací programy (unittest, pytest, nose2, ...)
  - Grafické programy (PyQT5, Tkinter, wxPython, ...)
  - Webové frameworky (Django, Flask, ...)

153

Patička souboru

```
if __name__ == "__main__":
    main()
```

- `main()` obvykle (variantně):
  - CLI aplikace
  - GUI aplikace
  - spuštění testů
  - spuštěné démona
  - spouštění paralelních vláken / procesů

- *What Does if __name__ == "__main__" Do in Python?*
https://realpython.com/if-name-main-python/

154

Struktury adresářů, projektů a balíčků

155

Projekt

- Obvyklé varianty výsledku:
  - Modul
  - Program, aplikace, skript
  - Knihovna, balíček
  - Spočtená data, grafy atd.
- Obvykle ve formě repozitáře na (GIT) serveru
- Doporučené uspořádání adresářů balíčku

```
Project/
├─ .gitignore
├─ requirements.txt
├─ README.md
├─ LICENSE
├─ pyproject.toml
├─ setup.py
├─ setup.cfg
├─ project/
│  ├─ __init__.py
│  ├─ main.py
│  ├─ … other code
│  └─ tests/
│     ├─ __init__.py
│     ├─ test_main.py
│     └─ … other tests
├─ doc/
└─ scripts/
   └─ project
```

Změnit na jméno projektu

156

Distribuce projektů a balíčků

- Distribuční balíčky:
  - built distribution = bdist (ready-to-install):
    - ~~egg (distribuce a balíčkování ; instalace runtimu)~~ ⇨
    - wheels (distribuce a balíčkování) (*.whl ~ *.zip)
  - source distribution = sdist (*.tar.gz)
- Konfigurační soubory pro distribuční balíčky:
  - ~~setup.py + setup.cfg~~ ⇨ pyproject.toml
- requirements.txt … pip install –r requirements.txt (vývoj)

158

Distribuce projektů a balíčků

- Při hledání informací, pozor na zastaralost údajů
- Python Packaging User Guide
https://packaging.python.org/en/latest/
- PyPA (Python Packaging Authority)
https://www.pypa.io/en/latest/
- setuptools (zdroj – PyPa)
https://setuptools.pypa.io/en/latest/index.html
- build (zdroj – PyPa)
https://build.pypa.io/en/stable/index.html
- Python Application Layouts: A Reference (zdroj – RealPython)
https://realpython.com/python-application-layouts/

159

Programátorské techniky (nejen) v Pythonu

160

Programátorské techniky

- Clean code / human-friendly code / readable code / managable code ⇨
- nikdo nenapíše napoprvé kód čistý (a obvbykel ani funkční) ⇨
- **refaktoring** (úprava kódu bez změny funkcionality) ⇨
- automatické testy (podporují kvalitu tím, že refaktoring nepřinese chyby)

161

„Velká" paradigmata programování

• Strukturované programování	≈Python standardně
• Imperativní programování	≈program = sekvence příkazů; akceptovatelní jen při testování, prototypování atd.
• Procedurální programování	≈program sekvence volání procedur
• Funkční programování	≈funkce `map()`, `reduce()`, ..., uzávěry [closure], lambda, ...
• (Deklarativní programování)	≈výjimečně

162

„Velká" paradigmata

• Objektové programování (OOP)	≈preferovaná technika
• Modulární programování	≈preferovaná technika
• Návrhové vzory [Design patterns]	
• Automatické testování	
• CI/CD = continuous integration / continuous delivery	

163

Principy návrhu kódu

- Vesměs stručné principy s dalekosáhlými důsledky
- **KISS** = **K**eep **I**t **S**imple, **S**tupid ⇨
(1) funkce/metody jsou malé
(2) srozumitelný kód bez „rafinovaností"
- **YAGNI** = **Y**ou **A**ren't **G**oing to **N**eed **I**t ⇨
  - neprogramovat funkcionalitu, kterou nepotřebujeme a možná (!!!) budeme potřebovat v budoucnu
  - žádné programování „pro strýčka Příhodu"

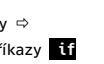
164

Principy návrhu kódu

- **DRY** = **D**on't **R**epeat **Y**ourself ⇨
  (1) podobný/stejný kód se neopakuje, ale je společnou funkcí/metodou/třídou
  (2) osobní/týmová/firemní knihovna znovupoužitelného kódu
- **DIE** = **D**uplication **I**s **E**vil ... cca stejné
- **WET** = **W**rite **E**verything **T**wice ... antipattern

165

Principy návrhu kódu



- **SESE** = **S**ingle **E**ntry, **S**ingle **E**xit = one return only ⇨
  - funkce/metoda více obsahuje vnořené příkazy `if`
  - cykly častěji obsahují `break`
  - funkce má lokální proměnou pro výsledek
  - return je poslední příkaz
- **Robustness** principle = liberalismus pro vstupní hodnoty (akceptujte libovolný vstup; zpracujte přijatelný vstup), konzervatismus pro výstup (výsledky přesném a v přesném formátu)

166

Principy návrhu kódu (OOP)

- OOP přináší další principy návrhu kódu včetně architektury
- **SRP** = **S**ingle **R**esponsibility **P**rinciple ⇨
  - funkce/metody/třídy dělají jen jednu jasně určenou věc ⇨
  - „vrstvení kódu" do řady malých komponent
- **CQS** = **C**ommand-**Q**uery **S**eparation ⇨
  funkce/metoda:
  (1) buď „něco počítá/dělá"
  (2) nebo vrací hodnotu (stav, výsledek)
  Pozn. obvykle vyžaduje požití OOP

167

Principy návrhu kódu (OOP)

- **SoC** = **S**eparation **o**f **C**oncernes ⇨
  - členit kód do menších částí (funkce / třídy /moduly)
  - omezovat vzájemné vazby

168

Principy návrhu kódu (OOP)

- **SOLID = SRP + OCP + LSP + ISP + DIP**
  - **SRP** = Single responsibility principle = funkce/metody/třídy dělají jen jednu jasně určenou věc
  - **OCP** = Open-closed principle = třída je otevřené k rozšíření/změně funkcionality (podtřídami; děděním), ale uzavřená pro nutnost změny základní třídy.
  - **LSP** = Liskov substitution principle = odkazy a reference na podtřídy lze při použití nahradit (substituovat) odkazy/referencemi na základní třídy

169

Principy návrhu kódu (OOP)

- **SOLID = SRP + OCP + LSP + ISP + DIP**
  - **ISP** = Interface segregation = Interface třídy (metody atd.) je oddělen od způsobu implementace. Třídy mají více malých oddělených rozhraní než jedno velké komplexní.
  - **DIP** = Dependency inversion = třídy mají být závislé na rozhraní používaných tříd/funkcí, ne na jejich implementaci.

170

Nevhodné techniky

- (antipatterns, smelling code)
- spaghetti code (dlouhé funkce/metody) ⇨ rozdělovat na funkce, délka cca 1÷2 obrazovky
- pizza code (příliš vnořených volání; příliš mnoho vrstev) ⇨ střídmě vytvářet „vrstvy" kódu
- složitost / komplexita kódu ⇨ rozdělovat na funkce/metody, délka cca 1÷2 obrazovky; příliš nevnořovat složené příkazy
- složitost / komplexita výpočtu ⇨ používat kvalitní algoritmy, používat ustálené knihovny, čerpat z kvalitní literatury

171

Nevhodné techniky

- používání jmen bez informační hodnoty ⇨ identifikátory s jasnou sémantikou podle jména; dodržovat jmenné konvence
- ignorování zavedených „best practices"⇨ používat kvalitní algoritmy, používat ustálené knihovny, čerpat z kvalitní literatury
- konstanty jsou skryté ⇨ konstanty uvádět v konfigurační části kódu nebo v konfiguračním souboru
- „znovuvynalézání kola"⇨ používat kvalitní algoritmy, používat ustálené knihovny, čerpat z kvalitní literatury

172

Nevhodné techniky

- nadměrné používání globálních proměnných ⇨ ideálně žádné globální proměnné; používat třídy
- zakomentovaný kód ⇨ nepoužívat, starý kód je ve verzovacím systému (GIT, ...)
- ladicí kód stále aktivní či jen zakomentovaný ⇨ důsledně mazat nebo konstrukce `if DEBUG: …`
- nejsou automatických testy ⇨ psát testy (`unittest`, `pytest`, `nose2`)

https://realpython.com/python-testing/

173

Nevhodné techniky

- neudržované komentáře neodpovídající kódu ⇨ udržovat komentáře; používat komentáře v omezené míře
- nadměrné používání nesrozumitelných „one-liners" ⇨ volání samostatných funkcí nebo použití složených příkazů
- copy/paste kódu bez porozumění ⇨ porozumět, testovat, popř. refaktoring
- AI vytvořený kód nekontrolovaný a bez porozumění ⇨ porozumět, testovat, popř. refaktoring

174

Nevhodné techniky

- výjimky jsou sice zachycené, ale nezpracované/ignorované⇨ zpracovávat výjimky, minimálně chybové hlášení (logování)
- výjimky jsou zachytávány všechny a bez rozlišení catch all ⇨ rozlišovat výjimky, použít web/AI/experimenty k nalezení vracených výjimek
- Ignorování návratových chybových stavů a výjimek ⇨ zpracovávat návratové kódy a výjimky
- používání/vytváření kódu a knihoven, které nehlásí chyby ⇨ zpracovávat a hlásit chybové stavy a výjimky

175

Nevhodné techniky

- … a další na individuální úrovni
- … a mnoho dalších při týmové práci
- … a mnoho dalších při řízení projektů

176

Refaktoring

- **Refaktoring identifikátorů / názvů** s cílem lepších identifikátorů a/nebo redukce komentářů
- **Refaktoring struktury příkazů** s cílem jednoduššího a srozumitelnějšího kódu
- **Refaktoring výpočetní logiky** s cílem jednoduššího, srozumitelnějšího nebo rychlejšího kódu
- **Refaktoring struktury tříd, funkcí a metod** s cílem menších tříd/metod/tříd , vyšší srozumitelnosti ...

177

Metodity vývoje a řízení

- Projektové řízení
- Vývoj software
- Agilní metodiky řízení
  - Scrum
  - Extreme Programming (XP)
  - Kanban
  - ... a mnoho dalších
- Agilní týmy/organizace
  - adaptace agilních metodik do velkých firem / projektů
  - Spotify metodika (squads / chapters / tribes / guild)

178

Programátorské techniky v Pythonu

179

Python techniky („Pythonish")

- pojmenovávat konstanty, popř. *.env
- type hinting
- výjimky jako podtypy standardních výjimek, nikoliv textové řetězce
- for count, item in enumerate (some_list)
- řetězení řetězců pomocí "".join(), nikoliv pomocí +
- regulární výrazy, přehledný zápis

180

Python techniky („Pythonish")

- typy Enum, Flag, ...
- sekvenční typy a jejich použití (list, tuple, UserList, namedtuple, kolekce, ...)
- typy pro pole/matice a jejich použití (list, tuple, dataframe, numpy array, numpy structured array)
- dekorátory funkcí, metod, tříd, ...
- Abstraktní datové typy collections.abc
- funkční programování
- Reprezentace obrázku

181

Python techniky („Pythonish")

- logging
- arg
- env
- I/O
- Closure
- Dekorátory
- Paralelní programování (vlákna, multiprocessing, asynchronní I/O, ...

182

Python OOP techniky („Pythonish“)

- OOP metodika
- veřejné, chráněné a privátní členy třídy
- dědění
- funkce super()
- @dataclass
- @classmethod
- @staticmethod
- @final
- @abstractmethod
- @property / @xxx.setter / @xxx.getter / @xxx.deleter
- magické metody („dunder“)

183

PEP 20 - The Zen of Python

- Principy psaní „pythonic“ kódu
- Velmi stručné, velmi hutné, často obtížné dodržet

- https://peps.python.org/pep-0020/
- https://pep20.org/

184

PEP 20 - The Zen of Python

- **Beautiful** is better than ugly.
- **Explicit** is better than implicit.
- **Simple** is better than complex.
- Complex is better than complicated.
- **Flat** is better than nested.

185

PEP 20 - The Zen of Python

- **Sparse** is better than dense.
- **Readability** counts.
- Special cases are **not special enough to break the rules**.
- Although **practicality** beats purity.
- **Errors should never pass silently**.
- Unless explicitly silenced.

186

PEP 20 - The Zen of Python

- In the face of ambiguity, refuse the temptation to guess.
- There should be one - and preferably only one - obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.

187

PEP 20 - The Zen of Python

- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the **implementation is easy to explain**, it may be a good idea.
- **Namespaces** are one honking great idea -- let's do more of those!

188

Nástroje

189

Doporučené nástroje

- Tzv. statická analýza kódu, tj. bez spuštění kódu
- Původně – samostatné nástroje, obvykle CLI (**C**ommand **L**ine **I**nterface = příkazová řádka)
- Moderně – integrace do grafických IDE (**I**ntegrated **D**evelopment **E**nvironmnet)
- Použití umělé inteligence (AI = **A**rtificial **I**ntelligence)

190

Python IDE

- **Visual Studio** Community/Professional/Enterprise (Microsoft)
- **Visual Studio Code** (Microsoft)
- **PyCharm** (JetBrains): Community/Professional
- **Jupyter**
- **Spyder**
- **PyDev**/LiClipse/Eclipse
- **PyScripter**
- Omezeně: editory (vim, Emacs, Atom, Sublime, Notepad++, …)
- Omezeně: (Python) IDLE

191

Výhody (Python) IDE

- Zvýraznění syntaxe
- Našeptávání/dokončování
- Ladění
- Statická analýza zobrazená v kódu
- Integrace GIT
- Integrace s AI
- Nástroje pro další jazyky a soubory (HTML, CSS, Markdown, make, shell, …)

192

Populární IDE a klávesové zkratky

Příkaz	PyCharm	Visual Studio Code
Všechny příkazy	Ctrl + Shift + A	Ctrl + Shift + P
Formátování kódu	Ctrl + Alt + L	Shift + Alt + F
„Zakomentování“ kódu	Ctrl + /	Ctrl + / Ctrl+ háček
Našeptávač	Ctrl + mezera	Ctrl + mezera
Refaktoring – přejmenuj identifikátoru	Shift + F6	F2
Refaktoring – vytvoř proměnnou	Ctrl + Alt + V	???
Refaktoring – vytvoř funkci/metodu	Ctrl + Alt + M	???
Run / continue	F9	F5
Step into	F7	F11
Step over	F8	F10

193

Nástroje umělé inteligence

194

Proč (učit) na ČVUT nástroje AI?

195

Základní problém s AI

- Otázky legality, autorství, plagiátorství, opisování ⇨

1. Vaše morální integrita
2. Firmy atd: ochrana duševního vlastnictví, ochrana dat, ochrana osobních dat, ...
3. Legislativa

196

AI - otázky legality a autorství

⇨ metodické pokyny ČVUT

- Metodický pokyn č. 5/2023 - Rámcová pravidla používání umělé inteligence na ČVUT pro studijní a pedagogické účely v Bc a NM studiu

https://www.cvut.cz/legislativa-tykajici-se-studia#umela-inteligence

- Metodický pokyn č. 2/2024 o dodržování etických principů při přípravě vysokoškolských závěrečných prací (zejména články 2.2, 2.5, 2.6)

https://www.cvut.cz/legislativa-tykajici-se-studia#eticke-principy

197

AI - otázky legality a autorství

...

2.2 nepřivlastňovat si cizí myšlenky a nápady či výsledky výzkumu bez uvedení jejich původního zdroje (nedopouštět se plagiátorství),

...

2.5 důsledně citovat autory, jejichž myšlenky nebo texty byly použity ve vlastní práci (při použití cizího díla ve větším rozsahu je vhodné si vyžádat i svolení autora),

2.6 uvést použití výstupů nástrojů umělé inteligence (dále jen UI),

...

198

Vlastnosti nástrojů AI

- Metody NLP, GPT, vesměs statistické textové modely !!!

⇨

- Někdy chybné až nesmyslné výsledky („halucinace").
- Nejasná autorská práva.
- Riziko krádeže obsahu (při dotazech a při integraci s IDE).
- Kvalita výsledku dána kvalitou vstupů a trénování modelů. Obvykle neznáte ani jedno.
- Kvalita výsledků dále dána postprocessingem (filtrací). Obvykle neznáte.

ALE

- Neustálý a bouřlivý rozvoj.

199

Doporučení k použití nástrojů AI

1. Používat !
2. Zohlednit pravidla ČVUT.
3. Přiznávat použití (použité zdroje, vygenerování kódu, ...).
4. Kvalifikační práce – doporučuji vést si podrobné záznamy o využití AI.
5. Nepoužívejte na kód a data, které nesmí opustit „počítač".

200

Doporučení k použití nástrojů AI

6. Nepoužívejte žádný výsledek AI, kterému nerozumíte.
7. Konzervatismus až nedůvěra v kvalitu výsledků z nástrojů umělé inteligence.
8. Kontrola správnosti odpovědí / výsledků.
9. Úprava podle vlastního úsudku a/nebo potřeb.
10. Popř. použít jen jako radu, inspiraci atd., ale nepoužívejte doslovně (tj. nepožívat „copy – paste“).

- Pozn. kód z GPT často používá příliš krátké identifikátory.

201

AI nástroje pro programování

- Pozn.: názvy technologií versus obchodní názvy produktů
- Pozn.: časté změny názvů z marketingových důvodů
- Velké množství
- Použití/generování:
  - Obecné texty
  - Obrázky
  - Zvuky/hudba
  - Videa
  - Specalizované texty, zejména programování

202

AI nástroje pro programování

- **OpenAI GPT-3.5**/GPT-4
- **MS Copilot**
- **GitHub CopilotX**, zdarma akademická licence
- **Tabnine** (0-12-39 USD/u/m)
- Amazon CodeWhisperer (0-19). Meta Code Llama,
- **Codeium** (free, $15/u/m)
- Google Duet AI for Developers = Gemini Code Assist = Bard ($19/u/m)
- GPT4All – open-source, offline
- tabby – open-source, offline

203

Microsoft Copilot

- Copilot Windows 11 ≠ Copilot 365 ≠ CopilotX = GitHub Copilot
- Využívá GPT-4
- Windows 11 Copilot (preview, rollout Sep23, zdarma)
  - vyžaduje Windows 11, 23H2, pravděpodobně též vyžaduej verzi EN a Ent/EDU
- Copilot web (https://copilot.microsoft.com/) / Bing search
  - GPT: Copilot/Designer/Vacation planner/Cooking ass./Fitnes trainer
- (GitHub) Copilot (X) … možnost akademické licence

204

Microsoft Copilot – další verze

- Copilot for Microsoft 365 (30$/u/m, min 300u, Office 365 vč. Teams)
- Copilot for Sales (2024)
- Copilot for Service (2024)
- Copilot in Viva
- Microsoft Security Copilot

205

Práce s AI nástroji

- Prompt = výzva = zadán pro práci
- Systém prompt
- API
- Context window

206

Možnosti AI při programování

- „Chytrý" vyhledávač (náhrada Google, DuckDuckGo, Bing, …)
- Dotazy v přirozeném jazyce
- Našeptávání [autocompletition]
- Generování kódu
- Generování testů
- Refaktoring kódu

207

Možnosti AI při programování

- Detekce a opravování chybv kódu
- Detekce chyb v bezpečnosti
- Tvorba dokumentace (Docstring i běžné komentáře)
- Vysvětlení kódu
- Type hinting
- Překlady jazyků (rodný jazyk ⇨ English)
- Vylepšení jazykových formulací („Czenglish" ⇨ English)

208

Způsoby použití AI

- Dotaz / odpověď na webu
- Dotaz / odpověď na webu v IDE
- Našeptávače IDE
- Generování kódu, komentářů atd. v IDE

209

AI – 1. dotaz/odpověď na webu

- Obecná webová rozhraní systémů GTP
  - Např: OpenAI GTP-3.5/GPT-4, MS Copilot, GitHub CopilotX, Meta Code Llama
  - Princip dotaz – odpověď, obvykle s udržením kontextu mezi dotazy
  - Zásadní „trik" – nastavení režimu práce tzv. promptem/výzvou

210

AI – 2. dotaz/odpověď v IDE

⇨ AI má plný přístup k Vašemu kódu a lze očekávat, že si ho stáhne a použije ke svému učení

- Obvykle vyžaduje instalaci doplňků (*pozn. situace jaro 2024*)
- Integrovaná webové rozhraní v IDE
  - Obvykle již přednastavený vhodný prompt (tj. režim práce)
  - Obvykle jazykové modely trénované jen nad relevantními texty
  - Integrace s editačním oknem

211

AI – 3. našeptávače

⇨ AI má plný přístup k Vašemu kódu a lze očekávat, že si ho stáhne a použije ke svému učení

- Obvykle nabízí IDE, též možnost jako instalaci doplňků (*pozn. situace jaro 2024*)
- Našeptávače (on-line navrhování kódu)
  - Dříve –podle pravidel jazyka a signatur tříd/metod/funkcí, resp. statisticky jaká slova byla použita v jiných programech (*Tabnine*)
  - Nově – návrhy i delších fragmentů kódu včetně použití správných již existujících proměnných

212

AI – 4. generování kódu v IDE

⇨ AI má plný přístup k Vašemu kódu a lze očekávat, že si ho stáhne a použije ke svému učení

- Obvykle nabízí IDE, též možnost jako instalaci doplňků (*pozn. situace jaro 2024*)
- Integrované nástroje v IDE (nativně nebo plugin)
  - generování kódu
  - generování Docstring
  - generování typehint
  - reformulace angličtiny
  - atd.

213

Další nástroje

214

Nástroje kontroly kódu

- Moderní speciální nástroje – **SonarQube / SonarLint**
- Statická kontrola kódu v IDE
- Klasické nástroje kontroly kódu – přístup přes IDE
- Klasické nástroje kontroly kódu – přístup přes CLI

215

Nástroje kontroly kódu

- PEP8 formátování / styl ➡ pycodestyle/pep8, **Flake8**
- Datové typy (*type checker*) ➡ **Mypy**, Pyright, Pytype
- Detekce chyby (*error linter*) ➡ **Pylint**, pyflakes, **Flake8**
- Nepoužitý / mrtvý kód ➡ Vultare, eradicate
- Komplexita / nepřehlednost ➡ McCabe, Radon
- Bezpečnost (hlavně web) ➡ Bezpečnost: Bandit

Nečistý kód (*code smells*)

216

Další nástroje

- Balíčkování (*packaging):* Pyroma

- Formátování kódu podle PEP8: black, autopep8
- Formátování Docstgring: pydocstringformatter, docformatter

217

Doporučené nástroje – pep8

- pycodestyle (= pep8) kontrola stylu podle PEP 8
- Integrován v PyCharm

- Dokumentace:
https://pycodestyle.pycqa.org/en/latest/intro.html

- Seznam pravidel/chyb – vhodné pro konfiguraci výjimek:
https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes
https://pypi.org/project/pep8-naming/

218

Doporučené nástroje – flake8

- flake8 (= PyFlakes + pycodestyle + McCabe)
  - Pycodestyle = pep8 = kontrola stylu
  - Pyflakes = kontrola syntaxe a hrubých chyb programování
  - McCabe = kontrola cyklomatické komplexity
- Dokumentace

https://media.readthedocs.org/pdf/flake8/latest/flake8.pdf

- Seznam pravidel/chyb – vhodné pro konfiguraci výjimek:

https://flake8.pycqa.org/en/latest/user/error-codes.html
https://www.flake8rules.com/

219

Doporučené nástroje

- Pylint
- MyPy
- SonarLint
- pydocstyle

- PyCharm & VisualStudioCode – nutné instalovat

220

Kvalita kódu

- **(Statická) složitost / komplexita** kódu jako míra pochopitelnosti, přehlednosti, udržovatelnosti ...

versus

- **Výpočetní složitost / komplexita**/náročnost – čas, paměť, O(n)

versus

- Složitost použitých (teoretických) metod, tj. matematika, fyzika, ...

221

Statická komplexita

- **Měření složitosti** s cílem získání metriky **pro zlepšování kódu**:
  - Cyklomatická komplexita
  - Počet řádek kódu
  - Počet řádek vykonatelného kódu
  - Halsteadovy index(y), speciálně Halsteadův objem
  - Index udržovatelnosti – kombinace výše uvedených
  - Kognitivní komplexita
- **Vysoká komplexita (obvykle) správně indikuje nepřehledný kód**.

222

Statická komplexita

- **Cyklomatická komplexita** *(Thomas McCabe, 1976. DoD)*
- Cca počítá definice funkcí/metod, větvení a cykly a jejich vnořování

- **Halsteadův objem** *(Maurice Howard Halstead,1977)*
- Cca počítá složitost výrazů a proměnných

- **Index udržovatelnosti** – kombinace výše uvedených plus počtu řádků

223

Komplexita / nepřehlednost

- **Kognitivní komplexita** *(G. Ann Campbell, 2016?, SonarSource)*
- Penalizuje:
  - narušení lineárního toku algoritmu (shora dolů, zprava doleva)
  - vnořené struktury
  - Tj. cca: `if`, `match`, `for`, `while`, `try-catch`, `break`, `continue`, rekurze, lambda funkce, složité logické výrazy, ternární operátory
- Nepenalizuje:
  - definice funkcí, metod a tříd
- Lépe vyhovuje moderním programovacím jazykům a moderním paradigmatům.

224

Řešení příliš vysoké komplexity

- **(Refaktoring identifikátorů / názvů)**
- **Refaktoring struktury příkazů :**
  - jednodušší podmínky (tj. vyčlenit výpočet podmínky do samostatného příkazu)
  - inverze podmínky
  - vnořené `if` ⇨ řetězec příkazů `if … elif … elif` …
  - vnořené `if` *nebo* řetězec `if … elif … elif`… ⇨ **příkaz** `match`

225

Řešení příliš vysoké komplexity

- **Refaktoring struktury funkcí a metod:**
  - vnořené cykly a podmínky ⇨ rozdělit na více menších funkcí/metod
  - dlouhý kód ⇨ rozdělit na více menších funkcí/metod/modulů
  - funkce/metody řízené (větvené) podle hodnot logických parametrů (tzv. příznaků) ⇨ použít dědění tříd; použít volání funkcí podle hodnoty příznaku
  - „včasné" příkazy `return`, je ale v rozporu s principem SESE
  - zredukovat duplicitní kód (viz princip DRY) ⇨ společný kód do parametrizované společné funkce/metody

226

Zdroje informací a sebevzdělávání

227

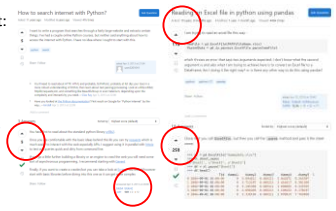
… zadarmo

- https://realpython.com/
- https://towardsdatascience.com/
- https://www.geeksforgeeks.org/
- https://www.digitalocean.com/community/
- https://datagy.io/
- https://docs.python.org/3/tutorial/index.html
- https://www.khanacademy.org/computing
- https://pypi.org/ …. najít balíček a k němu dokumentaci
- https://www.edx.org

228

… zadarmo

- Weby významných univerzit: MIT, Stanford, Harvard, Cambridge, Berkeley …
- https://stackoverflow.com/ … klasický zdroj informací pro programátory, pozor na kvalitu, pozor na stáří textu



229

… zadarmo nebo za málo

https://knihy.nic.cz/
https://greenteapress.com/wp/think-python/
https://github.com/pamoroso/free-python-books
https://pythonbooks.org/free-books/

- **Humble Bundle** – e-knihy renomovaných vydavatelství s výraznou slevou
https://www.humblebundle.com/

230