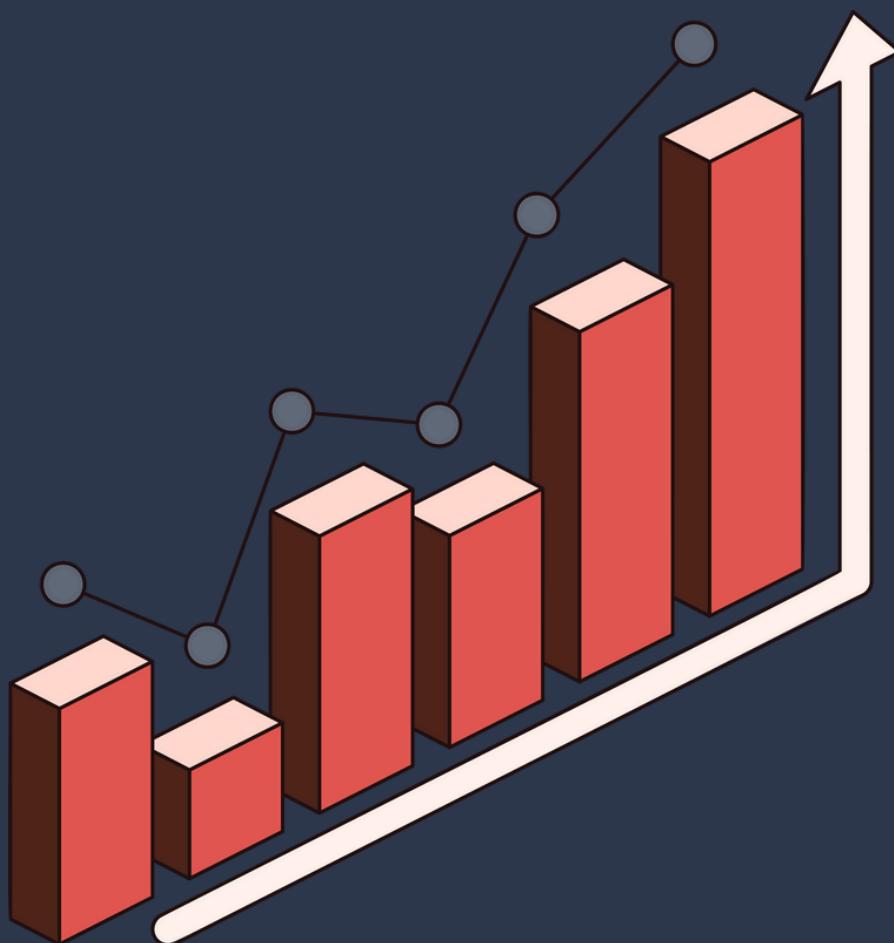


PYTHON SURVIVAL GUIDE



č. 1

VŠETKO, ČO POTREBUJEŠ NA PREŽITIE V PYTHONE
ČO TI NEPOVEDIA V ŠKOLE

RoboSkillz.Academy Číslo 1

Python Survival Guide

Všetko, čo potrebuješ na prežitie v Pythone

Čo ti v nepovedia v škole

Martin Hronský

Obsah

Sekcia I: Základy Pythonu

Úvod

Premenné a dátové typy

Pretypovanie a vstup od používateľa

Sekcia II: Riadiace štruktúry

Aritmetické operátory

Podmienky

Reálne projekty: Kalkulačka + Konverzie hmotnosti/teploty

Sekcia III: Logické operátory a práca s reťazcami

Logické operátory

Podmienené (ternárne) výrazy

Práca s reťazcami

Sekcia IV: Cykly

For cykly

While cykly

Vnorené cykly

Sekcia V: Dátové štruktúry

Listy (zoznamy)

Sety (množiny)

Tuple (n-tice)

Slovníky (dictionaries)

2D kolekcie

Sekcia VI: Náhodnosť a interaktivita v programovaní

Random

Sekcia VII: Tvorba funkcií a správa parametrov

Funkcie

Argumenty, *args, **kwargs

Iterables (iterovateľné objekty)

Sekcia VIII: Efektívna syntax

List comprehensions

Match-case

Moduly

Scope (lokálny a globálny)

Sekcia IX: Štruktúrovanie kódu a práca s hlavnými skriptmi

Main

Sekcia X: Objektovo orientované programovanie

Základy objektovo orientovaného programovania (OOP)

Premenné triedy

Delenie (inheritance)

Sekcia XI: Pokročilé OOP

Pokročilé OOP (polymorfizmus, duck typing)

Statické a triedne metódy

Magické metódy

Sekcia XII: Výnimky, súborový systém a dekorátory

@property

Dekorátory

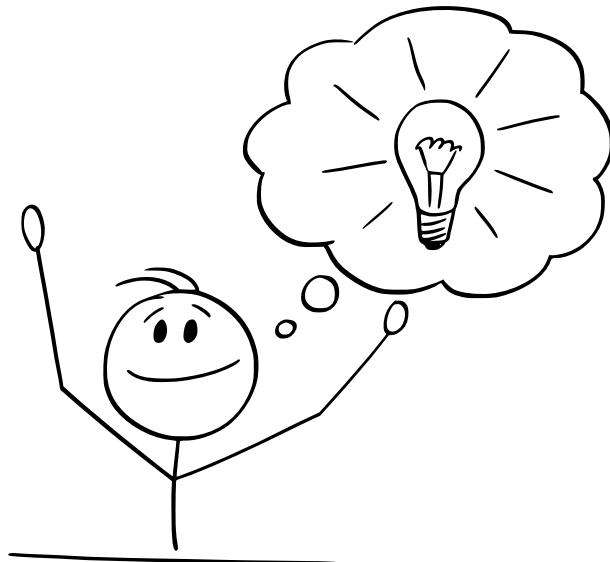
Výnimky (exception handling)

Práca so súbormi

Detekcia súborov

Úvod

"Pokiaľ si to dokážeš predstaviť, dať na papier, potom dokážeš naprogramovať čokoľvek."
- Motto mentorov RoboSkillz Academy



Väčšina ľudí, ktorí sa učia Python, zlyhá.

Nie preto, že by nemali talent.

Nie preto, že by nemali čas.

Zlyhajú preto, že sa učia spôsobom, ktorý nikdy nemôže fungovať.

Sledujú hodiny tutoriálov, ktoré im sľubujú, že sa stanú programátormi. Preklikávajú YouTube, čítajú náhodné články, skúšajú si útržky kódu, ktoré nerozumejú, a dúfajú, že to celé raz začne dávať zmysel.

Lenže nezmysel ostane nezmyslom.

Po týždňoch alebo mesiacoch sú frustrovaní, lebo aj keď si pamätajú zopár príkazov, nedokážu spraviť ani jednoduchú úlohu sami.

Táto kniha je iná.

Nie je to učebnica, ktorá sa tvári dôležito a ťa utopí v stovkách strán suchých teórií.

Nie je to ani zoznam príkazov, ktoré ti nikdy nikto neukáže, ako sa používajú v praxi.

A určite to nie je ďalší tutoriál, ktorý sa skončí práve vtedy, keď ho najviac potrebuješ.

Toto je **Survival Guide**.

Pretože na začiatku nejde o to, vedieť všetko.

Na začiatku ide o to **prežiť**.

Prežiť prvé hodiny programovania.

Prežiť maturitu z informatiky.

Prežiť pocit, keď sedíš pred prázdnym editorom a nemáš šajnu, kde začať.

Preto v tejto knihe dostaneš:

- **12 modulov základov Python**, od prvého príkazu až po základy objektovo orientovaného programovania.
- **Prvé projekty** - kalkulačka, hry, práca s textom, analýza dát.
- **Tipy a skratky mentorov**, ktoré ušetria hodiny trápenia.
- **Roadmapu**, ktorá ti ukáže, kam sa máš pohnúť ďalej.
- **BONUS**: Zľavu na kompletný kurz Python 1.0 - Základy, ako moju vdăku za čítanie tejto knihy.

Je to tvoje minimum na prežitie.

S týmto zvládneš napísat svoje prvé programy.

A keď zvládneš prvé programy, zvládneš aj projekty.

A keď zvládneš projekty, zistíš, že Python ti otvára dvere, o ktorých si si myslel, že sú zatvorené.

Ale jedna vec je jasná:

Toto nie je celý kurz.

Tu dostaneš jadro, všetko podstatné, čo potrebuješ, aby si sa prestal trápiť a začal tvoriť.

Ale ak to myslíš vážne, čakajú ťa ešte **bonusy, Real-Life Lab výzvy, Director's Cut videá, diskusie, cvičenia a mentoring** – to všetko nájdeš v plnom kurze.

Táto kniha ti dá zbraň.

Kurz ti dá výcvik.

Takže tu je dohoda:

Ak otvoríš tieto strany, odídeš s istotou, že **Python pre teba nie je strašiak, ale nástroj, ktorý vieš ovládať**.

My ti dáme cestu.

Ty musíš urobiť prvý krok.

Tak začnime.

Sekcia I: Základy Pythonu

Základné príkazy v Pythone

`print()` – prvý príkaz, ktorý musíš poznať

V Pythone sa na výpis do konzoly používa funkcia `print()`.

Čokoľvek vložíš do zátvoriek, Python zobrazí na obrazovke.

Príklady:

```
print("Hello World!")
print(42)
print(3.14)
```

Môžeš spájať text a čísla:

```
meno = "Marek"
vek = 17
print("Mám", vek, "rokov a volám sa", meno)
```

Od verzie Python 3.6 vieš použiť aj **f-stringy**:

```
print(f"Ahoj, volám sa {meno} a mám {vek} rokov.")
```

Ja viem, je toho naraz veľa, k všetkému sa dostaneme a postupne si jednotlivé veci vysvetlíme.

Komentáre – text, ktorý Python ignoruje

Komentár je poznámka v kóde, ktorú programátor píše pre seba alebo iných. Python ich **nevýkonáva**.

Jednoriadkový komentár

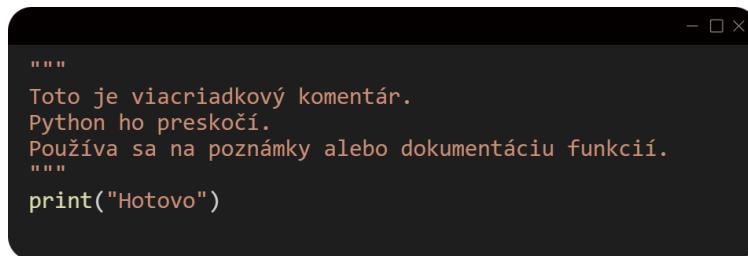
Začína sa znakom `#`.

```
# Toto je komentár
print("Ahoj") # komentár môže byť aj za príkazom
```

Viacriadkový komentár

Použi trojité úvodzovky (""" alebo ''').

Vhodné na dlhšie vysvetlenia alebo dokumentáciu.



```
"""
Toto je viacriadkový komentár.
Python ho preskočí.
Používa sa na poznámky alebo dokumentáciu funkcií.
"""
print("Hotovo")
```

Prečo sú komentáre dôležité

- Pomáhajú **pochopíť logiku** programu aj po čase.
- Uľahčujú prácu v tíme.
- Učiteľ/mentor rýchlo zistí, čo má kód robiť.

Začiatočníci robia chybu, že komentujú úplne všetko (napr. `# nastavím premennú x = 5`).

Správne komentáre vysvetľujú **prečo**, nie **čo**.

Mini úloha

1. Vypíš svoj oblúbený citát cez `print()`.
2. Pridaj nad to komentár s vysvetlením, prečo si ho vybral.
3. Skús napísať viacriadkový komentár o sebe a potom vypíš svoje meno a vek.

Premenné a pretypovanie (Variables & Type casting)

Čo sú premenné (a prečo na nich záleží)

Premenná je **menovka na hodnotu v pamäti**. Názvom sa na hodnotu odkazuješ, meníš ju, používaš v ďalších výpočtoch.

Premennú si predstav ako kufrík, do ktorého si vieme niečo odložiť. Keď cestuješ na dovolenku, pravdepodobne si zbalíš plavky a opaľovací krém, do **premennej** si pri programovaní odložíme číslo, text, alebo iný druh hodnoty. Kedykoľvek tento kufrík viem otvoriť a vložiť doň nový druh hodnoty. Má to ale pár pravidiel, ktoré sú popísne nižšie.

Zásady:

- Názvy: malé písmená, **snake_case** → **pocet_bodov**, **priemer**, **meno_studenta**.
- Vyhni sa kľúčovým slovám (**for**, **if**, **class**...).
- Názov nech vyjadruje **zmysel**, nie typ (**vek** lepšie než **x**).

Napríklad:

```
vek = 17          # int
priemer = 1.82    # float
meno = "Jana"     # str (string)
je_plnolety = False # bool
```

Základné dátové typy

- **int** – celé číslo: -2, 0, 42
- **float** – desatinné: 3.14, -0.5
- **str (string)** – text: "Python", 'Ahoj'
- **bool** – logická hodnota: True, False

Typ hodnoty zistíš:

```
print(type(42))      # <class 'int'>
print(type(3.14))     # <class 'float'>
print(type("Python"))  # <class 'str'>
print(type(True))      # <class 'bool'>
```

Pretypovanie (type casting)

Pretypovanie = **vytvorenie novej hodnoty iného typu** zo starej.

- `int(x)` → pokus premeniť na celé číslo
- `float(x)` → pokus premeniť na desatinné
- `str(x)` → textová reprezentácia

Príklady:

```
a = "42"
b = int(a)          # 42 (int)

c = "3.14"
d = float(c)        # 3.14 (float)

e = 7
f = str(e)          # "7" (str)
```

Pozor (typické chyby):

```
int("3.14") # ValueError (string s bodkou nie je celé číslo)
float("abc") # ValueError (ved' z abc nevieme urobiť True/False)
```

Pretypovanie a vstup od používateľa

Na to, aby sme neustále nemuseli prepisovať program a meniť jeho hodnoty, vieme využiť funkciu `input()`. Táto funkcia do termínálu vypíše čokoľvek, čo sa nachádza medzi jej zátvorkami, a následne bude čakať, kým ty sám nedopíšeš do konzoly svoju odpoveď.

TIP: `input()` vždy vracia string. Ak chceš počítať s číslami, musíš pretypovať.

```
vek = int(input("Zadaj svoj vek: "))      # "17" -> 17
vyska = float(input("Zadaj výšku (m): ")) # "1.75" -> 1.75
print("O rok budeš mať", vek + 1)
```

Na začiatku si počítač vypýta od teba v konzole, aby si zadal svoj vek (napísali sme 17). Lenže táto hodnota je string, pretože `input()` vracia len string. Pokiaľ chceme počítať s hodnotou 17, musíme ju pretypovať na číslo cez `int()` a potom uložiť do premennej `vek`.

To isté sa deje aj s výškou, avšak výška je float, pretože moja výška je 175 cm, v skratke 1.75 metra.

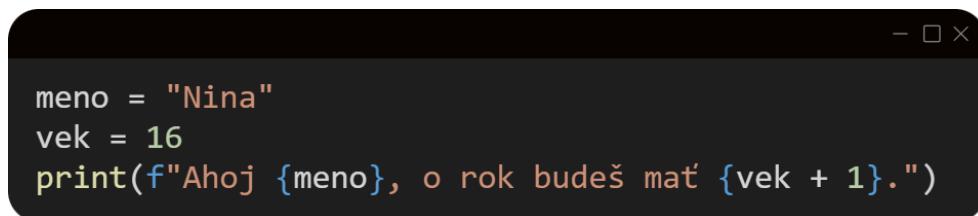
Na záver vypíšem do konzoly vetu "O rok budeš mať" a k tomu za pomoci čiarky pridám vek a zvýšim ho o jedna (keď mám teraz s 17, tak aby som o rok mal 18). Keby bol vek stále string a my sme ho predtým nepretypovali, nastal by error.

Formátovanie výstupu (f-stringy)

Najpohodlnejší spôsob, ako spájať text a premenné.

F-string je formát, ktorým viem upravovať text. Všimite si malého `f` pred úvodzovkami hore: `f"Ahoj {meno}, o rok budeš mať {vek + 1}."`

Vďaka tomu malému `f` nepotrebuje pridať čiarky ani znaky `+` na spájanie textov, dokonca ani pretypovať nemusíme, program si natiahne hodnoty podľa toho, aká premenná sa nachádza v zátvorkách `{}`.



```
meno = "Nina"
vek = 16
print(f"Ahoj {meno}, o rok budeš mať {vek + 1}.")
```

TIP:

- **Získanie dĺžky reťazca:** `len("Python") # 6`
 - **Konverzia textu na veľké/malé:**
`"ahoj".upper() → "AHOJ", "Ahoj".lower() → "ahoj"`
 - **Zlepenie textu:** `"Ahoj " + meno alebo f"Ahoj {meno}"`
-

Časté chyby (a rýchle opravy)

NESPRÁVNE:

```
vek = input("Vek: ")
print(vek + 1)
```

SPRÁVNE:

```
vek = int(input("Vek: "))
print(vek + 1)
```

NESPRÁVNE: `int("3.14")`

SPRÁVNE: `float("3.14")` alebo najprv `float(...)`, potom `int(...)` ak chceš odrezať desatinnú časť:
`int(float("3.14"))`

Mini-projekt „Madlibs“ (premenné + input + f-stringy)

Zadáš pári slov a poskladáš z nich krátkejší príbeh.

- □ ×

```
meno = input("Zadaj meno: ")
miesto = input("Meno mesta: ")
aktivita = input("Oblúbená aktivita: ")
cislo = int(input("Oblúbené číslo: "))

pribeh = (
    f"{meno} dnes odišiel do mesta {miesto}. "
    f"Po ceste si dal {cislo} prestávok a venoval sa aktivite: {aktivita}. "
    "Bol to super deň!"
)
print(pribeh)
```

Sekcia II: Riadiace štruktúry

Aritmetické operátory

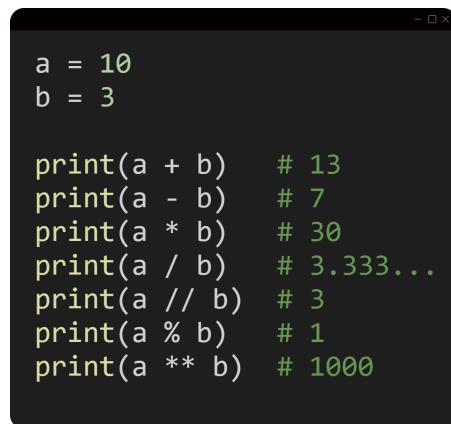
Čo sú aritmetické operátory

V Pythone (ako v matematike) vieš s číslami počítať pomocou operátorov.

Najčastejšie sú:

- `+` sčítanie
- `-` odčítanie
- `*` násobenie
- `/` delenie (výsledok je vždy **float**)
- `//` celočíselné delenie (odrezáva desatinnú časť)
- `%` modulo (zvyšok po delení)
- `**` mocnina

Príklady



```
a = 10
b = 3

print(a + b)      # 13
print(a - b)      # 7
print(a * b)      # 30
print(a / b)       # 3.333...
print(a // b)      # 3
print(a % b)       # 1
print(a ** b)      # 1000
```

Porovnanie / matematické pravidlá

Priorita operátorov je ako v matematike:

1. mocniny
2. násobenie/delenie
3. sčítanie/odčítanie.

Na zmenu poradia použi **zátvorky**.



```
print(2 + 3 * 4)      # 14
print((2 + 3) * 4)    # 20
```

Typické chyby

1. Zamieňanie / a //

```
print(5 / 2)    # 2.5
print(5 // 2)   # 2
```

1. Modulo nie je delenie

```
print(10 % 3)  # 1 (zvyšok po delení)
```

1. Mocnina nie je ^

```
print(2 ^ 3)   # 1 (XOR, nie mocnina!)
print(2 ** 3)  # 8 (správne)
```

Mini-cvičenia

1. Spočítaj, koľko celých hodín a koľko minút ostane z **135 minút**.
2. Zisti výsledok **17 // 4** a **17 % 4**.
3. Vypočítaj obsah obdĺžnika so stranami **8 × 5**.
4. Skús **2 ^ 5** a čo sa stane? Ako to opraviť?

If podmienky

Prečo If podmienky?

Doteraz sme vedeli len **vypočítať výsledok**. Ale program, ktorý nevie rozhodovať, je ako kalkulačka bez tlačidla „=“.

Pomocou **if** dávame programu schopnosť **reagovať na rôzne situácie**.

Porovnávacie operátory

- **==** rovnosť
- **!=** nerovnosť
- **>** väčšie
- **<** menšie
- **>=** väčšie alebo rovné
- **<=** menšie alebo rovné

V Pythone vždy dostaneš **True** alebo **False**.

Základná štruktúra

```
- □ ×  
if podmienka: # toto je hlavička if podmienky  
    """ tu je kód, ktorý sa vykoná,  
    ak je podmienka True"""
```

Príklad:

```
- □ ×  
vek = 18  
if vek >= 18:  
    print("Si plnoletý.")
```

Všimnite si, že kód pod hlavičkou je posunutý vpravo (štandardne cez tlačidlo TAB alebo 4 medzery). Vďaka tomu program vie, ktoré riadky sa týkajú podmienky, a ktoré už nie.

If – Else

else sa používa v prípade, pokiaľ chcem ošetriť aj možnosť, že podmienka výjde **False**.

```
if podmienka:  
    # ak je True  
else:  
    # ak je False
```

Príklad:

```
cislo = 7  
if cislo % 2 == 0:  
    print("Číslo je párne.")  
else:  
    print("Číslo je nepárne.")
```

Z predošej témy vieme, že znak `%` je delenie bezozvyšku. Keď je číslo nepárne, jeho výsledok po delení 2 bude vždy 1 a pri párnych bude výsledok 0 (pretože 5/2 je 0 zv. 1 a 4/2 je 2 zv. 0).

If – Elif – Else

Používa sa, ak máš viac možností.

```
znamka = 2  
  
if znamka == 1:  
    print("Výborný")  
elif znamka == 2:  
    print("Chválitebný")  
elif znamka == 3:  
    print("Dobrý")  
else:  
    print("Treba zabratť")
```

Typické chyby

Zámena `=` a `==`

```
if x = 5:    # chyba!
```

Správne:

```
if x == 5:
```

Zabudnutá dvojbodka

```
if x > 0    # SyntaxError
```

Nesprávne odsadenie (chýbajú medzery naľavo od `print()`)

```
if x > 0:  
print("kladné")  # IndentationError
```

Mini-cvičenia

1. Napíš program, ktorý sa spýta na vek a vypíše, či je človek plnoletý.
2. Spýtaj sa na číslo a zistí, či je párne alebo nepárne.
3. Program, ktorý vyhodnotí známku (1–5) a vypíše text (ako v príklade vyššie).
4. Zistí, či zadané číslo je väčšie, menšie alebo rovné nule.

Reálne projekty

Program Kalkulačka

Ciel: Používateľ zadá dve čísla a operáciu, potom program vypočíta výsledok.

```
# Jednoduchá kalkulačka
a = float(input("Zadaj prvé číslo: "))
b = float(input("Zadaj druhé číslo: "))
operacia = input("Zadaj operáciu (+, -, *, /): ")

if operacia == "+":
    print(f"Výsledok: {a + b}")
elif operacia == "-":
    print(f"Výsledok: {a - b}")
elif operacia == "*":
    print(f"Výsledok: {a * b}")
elif operacia == "/":
    if b != 0:
        print(f"Výsledok: {a / b}")
    else:
        print("Chyba: delenie nulou nie je možné")
else:
    print("Neznáma operácia")
```

Typické chyby pri kalkulačke

1. Zabudnuté pretypovanie → všetko z `input()` je string.
2. Zabudnutá kontrola delenia nulou.
3. Operátor, ktorý program nepozná → treba mať vetvu `else`.

Mini cvičenia ku Kalkulačke

1. Rozšír kalkulačku o **mocninu**.
2. Rozšír kalkulačku o **odmocninu**.

Konverzie hmotnosti a teploty

Ciel: Prevod medzi dvoma jednotkami podľa výberu používateľa.

Hmotnosť (kg ↔ lbs)

```
hmotnost = float(input("Zadaj hmotnosť: "))
jednotka = input("Jednotka (kg alebo lbs): ")

if jednotka == "kg":
    print(f"{hmotnost} kg = {hmotnost * 2.20462:.2f} lbs")
elif jednotka == "lbs":
    print(f"{hmotnost} lbs = {hmotnost / 2.20462:.2f} kg")
else:
    print("Neznáma jednotka")
```

Teplota ($^{\circ}\text{C} \leftrightarrow ^{\circ}\text{F}$)

```
teplota = float(input("Zadaj teplotu: "))
jednotka = input("Jednotka (\text{C alebo F}): ")

if jednotka.upper() == "C":
    print(f"{teplota} \text{ } ^{\circ}\text{C} = {(teplota * 9/5) + 32:.2f} \text{ } ^{\circ}\text{F}")
elif jednotka.upper() == "F":
    print(f"{teplota} \text{ } ^{\circ}\text{F} = {(teplota - 32) * 5/9:.2f} \text{ } ^{\circ}\text{C}")
else:
    print("Neznáma jednotka")
```

Mini cvičenia ku Konverziám

1. Rozšír program o možnosť previesť **minúty** na **hodiny a minúty**.
2. Dopíš funkciu na prevod **eur** na **doláre** (menový kurz si zadaj ako premennú).
3. Napíš program, ktorý sa spýta, akú konverziu chce používateľ (hmotnosť alebo teplota), a spustí správny blok kódu.

Sekcia III: Logické operátory a práca s reťazcami

Logické operátory

Prečo logické operátory?

If podmienky zo Sekcie II nám umožnili robiť rozhodnutia v programe.

Ale čo ak chceme skombinovať viacero podmienok naraz?

Tu prichádzajú na rad logické operátory.

Logické operátory v Pythone

- **and** – platí len vtedy, keď sú obe podmienky True
- **or** – platí, keď aspoň jedna podmienka je True
- **not** – negácia, otočí True ↔ False

Príklady:

```
vek = 18
student = True

# and
if vek >= 18 and student:
    print("Plnoletý študent")

# or
if vek < 18 or student:
    print("Buď je mladší, alebo je študent")

# not
if not vek < 18:
    print("Nie je mladší ako 18")
```

Podmienené (ternárne) výrazy

Niekedy chceme jednoduchú vetu **if/else** zapísť **na jeden riadok**.

Syntax:

```
výraz1 if podmienka else výraz2
```

Príklad:

```
vek = 20
status = "plnoletý" if vek >= 18 else "neplnoletý"
print(status)    # plnoletý
```

Typické chyby

Použitie & namiesto and

NESPRÁVNE:

```
if x > 0 & y > 0:    # zlé, je to bitový operátor
```

SPRÁVNE:

```
if x > 0 and y > 0:
```

Zámena logiky

NESPRÁVNE:

```
if not x > 0:    # ťažko čitateľné
```

SPRÁVNE:

```
if x <= 0:
```

Mini cvičenia

1. Program, ktorý zistuje, či je číslo v rozsahu 1–10 (vrátane).
2. Spýtaj sa používateľa na vek a či je študent. Vypíš „Zľava platí“, ak má menej ako 26 a zároveň je študent.
3. Spýtaj sa na heslo. Ak je správne, vypíš „Vitaj“. Ak nie, vypíš „Zlý pokus“.
4. Prepíš úlohu z bodu 3 do formy **ternárneho výrazu**.

Práca s reťazcami

Čo je string (reťazec)

String = text, ktorý dávame do **úvodzoviek** ("... alebo '...').

Každý znak má svoju **pozíciu (index)** → Python vie s textom pracovať ako so zoznamom znakov.

Prvý znak nejakého textu je vždy **indexu 0**. Je to tak neutrálne vo všetkých jazykoch.

Posledný znak vieme dostať (bez toho, aby som potreboval poznať dĺžku textu...lebo nie vždy budeme dĺžku textu poznať) z **indexu -1**.

Príklad:

```
- □ ×  
slovo = "Python"  
print(slovo[0])    # P  
print(slovo[1])    # y  
print(slovo[-1])   # n (posledný znak)
```

Užitočné string metódy

len() – dĺžka textu

```
- □ ×  
meno = "Nina"  
print(len(meno))    # 4
```

count() – počet výskytov v danom teste

```
- □ ×  
text = "banana"  
print(text.count("a"))    # 3
```

Syntax je vždy: *názov premennej, v ktorej sa nachádza daný reťazec (v tomto prípade „text“), potom .count() a do zátvoriek vždy ide text, ktorého počet chceme nájsť (v tomto prípade „a“).*

"xyz" in text – obsahuje/ neobsahuje

```
- □ ×  
text = "Python je super"  
print("Python" in text)    # True  
print("Java" not in text)  # True
```

`lower()` a `upper()` – zmena veľkých/malých písmen

```
text = "Ahoj"  
print(text.lower())    # ahoj  
print(text.upper())    # AHOJ
```

`find()` – pozícia podreťazca (alebo -1, ak sa nenašiel)

```
text = "Banán"  
print(text.find("n"))    # 2  
print(text.find("x"))    # -1
```

`index()` – pozícia podreťazca (podobné ako `find()` ked' sa nenašiel, vyhodí chybu)

`replace()` – nahradenie časti textu

```
text = "Ahoj svet"  
novy = text.replace("svet", "Python")  
print(novy)    # Ahoj Python
```

Znakové kódy (`ord()` a `chr()`)

Každý znak má svoje číslo v ASCII/Unicode.

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0 000	NUL	(null)	32	20 040	 	Space		64	40 100	@	Ø	96	60 140	`	`		
1	1 001	SOH	(start of heading)	33	21 041	!	!	!	65	41 101	A	A	97	61 141	a	a		
2	2 002	STX	(start of text)	34	22 042	"	"	"	66	42 102	B	B	98	62 142	b	b		
3	3 003	ETX	(end of text)	35	23 043	#	#	#	67	43 103	C	C	99	63 143	c	c		
4	4 004	EOT	(end of transmission)	36	24 044	$	\$	\$	68	44 104	D	D	100	64 144	d	d		
5	5 005	ENQ	(enquiry)	37	25 045	%	%	%	69	45 105	E	E	101	65 145	e	e		
6	6 006	ACK	(acknowledge)	38	26 046	&	&	&	70	46 106	F	F	102	66 146	f	f		
7	7 007	BEL	(bell)	39	27 047	'	'	'	71	47 107	G	G	103	67 147	g	g		
8	8 010	BS	(backspace)	40	28 050	(((72	48 110	H	H	104	68 150	h	h		
9	9 011	TAB	(horizontal tab)	41	29 051)))	73	49 111	I	I	105	69 151	i	i		
10	A 012	LF	(NL line feed, new line)	42	2A 052	*	*	*	74	4A 112	J	J	106	6A 152	j	j		
11	B 013	VT	(vertical tab)	43	2B 053	+	+	+	75	4B 113	K	K	107	6B 153	k	k		
12	C 014	FF	(NP form feed, new page)	44	2C 054	,	,	,	76	4C 114	L	L	108	6C 154	l	l		
13	D 015	CR	(carriage return)	45	2D 055	-	-	-	77	4D 115	M	M	109	6D 155	m	m		
14	E 016	SO	(shift out)	46	2E 056	.	.	.	78	4E 116	N	N	110	6E 156	n	n		
15	F 017	SI	(shift in)	47	2F 057	/	/	/	79	4F 117	O	O	111	6F 157	o	o		
16	10 020	DLE	(data link escape)	48	30 060	0	0	0	80	50 120	P	P	112	70 160	p	p		
17	11 021	DC1	(device control 1)	49	31 061	1	1	1	81	51 121	Q	Q	113	71 161	q	q		
18	12 022	DC2	(device control 2)	50	32 062	2	2	2	82	52 122	R	R	114	72 162	r	r		
19	13 023	DC3	(device control 3)	51	33 063	3	3	3	83	53 123	S	S	115	73 163	s	s		
20	14 024	DC4	(device control 4)	52	34 064	4	4	4	84	54 124	T	T	116	74 164	t	t		
21	15 025	NAK	(negative acknowledge)	53	35 065	5	5	5	85	55 125	U	U	117	75 165	u	u		
22	16 026	SYN	(synchronous idle)	54	36 066	6	6	6	86	56 126	V	V	118	76 166	v	v		
23	17 027	ETB	(end of trans. block)	55	37 067	7	7	7	87	57 127	W	W	119	77 167	w	w		
24	18 030	CAN	(cancel)	56	38 070	8	8	88	58 130	X	X	120	78 170	x	x			
25	19 031	EM	(end of medium)	57	39 071	9	9	9	89	59 131	Y	Y	121	79 171	y	y		
26	1A 032	SUB	(substitute)	58	3A 072	:	:	:	90	5A 132	Z	Z	122	7A 172	z	z		
27	1B 033	ESC	(escape)	59	3B 073	;	:	:	91	5B 133	[[123	7B 173	{	{		
28	1C 034	FS	(file separator)	60	3C 074	<	<	<	92	5C 134	\	\	124	7C 174	|			
29	1D 035	GS	(group separator)	61	3D 075	=	=	=	93	5D 135]]	125	7D 175	}	}		
30	1E 036	RS	(record separator)	62	3E 076	>	>	>	94	5E 136	^	^	126	7E 176	~	~		
31	1F 037	US	(unit separator)	63	3F 077	?	?	?	95	5F 137	_	_	127	7F 177		DEL		

Source: www.LookupTables.com

- ord(znak) → číslo
- chr(číslo) → znak

```
print(ord("A"))    # 65
print(chr(65))    # A
```

Používa sa to napríklad pri šifrovaní a my to použijeme neskôr pri Cézarovej šifre.

Cézarova šifra (mini projekt)

Jednoduchá šifra: každý znak posunieme o n pozícii v abecede.

```
text = "abc"
posun = 3
zasifrovane = ""

for znak in text:
    zasifrovane += chr(ord(znak) + posun)

print(zasifrovane) # def
```

Problémy na premýšľanie:

- čo ak posun „prelezie“ za `z`?
- čo s medzerami alebo číslicami?

Indexovanie reťazcov

- **Prvý znak:** `text[0]`
- **Od indexu 0 po 2:** `text[0:3]` (protože druhé číslo je otvorený interval <0;3), teda posledný znak sa nečíta)
- **Každý druhý znak:** `text[::-2]`
- **Text od zadu:** `text[::-1]`

```
text = "Python"
print(text[0:3]) # Pyt
print(text[::-1]) # nohtyP
```

Jednoduchý program na úpravu textu

Úloha: Používateľ zadá text a program vypíše základné štatistiky.

```
text = input("Napíš text: ")

print("Dĺžka textu:", len(text))
print("Počet písmen a:", text.count("a"))
print("Malými písmenami:", text.lower())
print("Veľkými písmenami:", text.upper())
print("Otočený text:", text[::-1])
```

Časté chyby

Index mimo rozsahu

```
slovo = "Python"  
print(slovo[10])    # IndexError
```

Rozdiel medzi `find()` a `index()`

- `find()` → -1 ak nenájde
- `index()` → vyhodí chybu

Nepoužívanie `str()` pri spojení textu a čísla

```
vek = 17  
print("Mám " + vek)      # TypeError  
print("Mám " + str(vek))# OK
```

Mini cvičenia

1. Vypíš, kolko krát sa v slove „programovanie“ nachádza písmeno a.
2. Napíš program, ktorý zistí, či text obsahuje slovo „Python“.
3. Vytvor program, ktorý prečíta meno a vypíše ho veľkými písmenami.
4. Skús otočiť text odzadu (`[::-1]`).
5. Bonus: Skús spraviť malý šifrovací program, ktorý posunie každý znak o 1 (Cézar).

Sekcia IV: Cykly

For cykly

Prečo potrebujeme cykly?

Predstav si, že máš vypísať čísla od 1 do 100. Napísať 100-krát `print()` by bol nezmysel. Tu prichádzajú **cykly** – umožnia nám opakovať kód **automaticky**.

Základná štruktúra for cyklu

Syntax:

```
for premenná in kolekcia: # hlavička for cyklu  
    # blok kódu, ktorý sa opakuje
```

- `kolekcia` = niečo, cez čo sa dá prechádzať (napr. zoznam, string, range).
- `premenná` = aktuálna hodnota pri každej iterácii.

Príklady s `range()`

Jednoduchý rozsah

```
for i in range(5):  
    print(i)  
# 0, 1, 2, 3, 4
```

Od – Do

```
for i in range(1, 6):  
    print(i)  
# 1, 2, 3, 4, 5
```

Tu môžeme vidieť, že sa nám hlavička cyklu trochu zmenila. Rovnako ako pri reťazcoch, tak aj tu platí, že interval je sprava otvorený, teda <1;6> sú čísla 1,2,3,4,5.

Skoky

```
for i in range(0, 10, 2):
    print(i)
# 0, 2, 4, 6, 8
```

Hlavička sa opäť trochu zmenila a pribudlo tretie číslo. Toto číslo popisuje veľkosť kroku do ďalšieho čísla/znaku, s ktorým budeme pracovať. 2 znamená každé druhé číslo/znak, 5 by znamenalo každé piaté.

Prechádzanie cez string

```
text = "Python"
for znak in text:
    print(znak)
# P, y, t, h, o, n
```

Prechádzanie cez zoznam

```
znamky = [1, 2, 1, 3, 2]
for z in znamky:
    print(z)
# 1, 2, 1, 3, 2
```

K zoznamom a ich použití sa dostaneme v neskorších sekciách, ale je dobré ich tu spomenúť.

enumerate() – index + hodnota

Niekedy chceme vedieť aj pozíciu (uvedený príklad sa dá použiť aj pre zoznamy).

```
ovocie = "noc"
for i, znak in enumerate(ovocie):
    print(i, znak)
# 0 n
# 1 o
# 2 c
```

Typické chyby

Zlá identácia (odsadenie)

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

Nesprávne parametre range - prvé číslo musí byť vždy menšie ako druhé

```
for i in range(5, 1):  # nič nevypíše  
    print(i)
```

Mini cvičenia

1. Vypíš čísla od 1 do 10.
2. Vypíš len párne čísla od 0 do 20.
3. Vypíš znaky textu "RoboSkillz".
4. Spočítaj, kol'ko je v texte "programovanie" písmen a.
5. Vypíš tabuľku násobilky pre číslo 7.

Miniprojekt: Hviezdičkové obrazce

Trojuholník

```
for i in range(1, 6):  
    print("*" * i)
```

Výstup:

```
*  
**  
***  
****  
*****
```

Štvorček

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

While cykly

Prečo while cykly?

for cyklus je skvelý, keď **vieš dopredu, kol'kokrát** sa má niečo opakovať (napr. čísla 1–10).

Ale čo ak **nevieš presne, kedy sa má cyklus zastaviť**?

Napríklad:

- opakuj, kým používateľ nenapíše správne heslo,
- počítaj, kým hodnota neklesne pod nulu,
- generuj, kým nedostaneš požadovaný výsledok.

Na to je tu **while**.

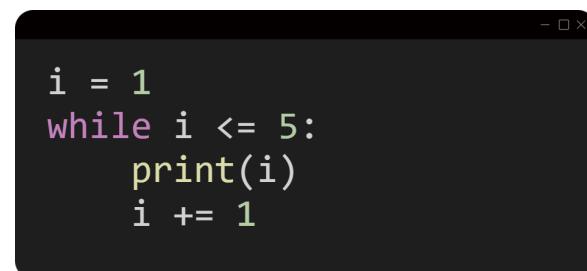
Základná štruktúra



```
while podmienka: # hlavička while cyklu
    # blok kódu
```

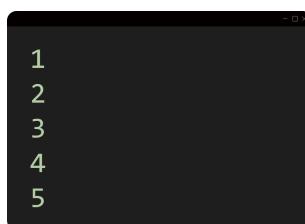
Kód sa vykonáva **tak dlho, kým je podmienka True**.

Jednoduchý príklad



```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Výstup:



```
1
2
3
4
5
```

Typické využitie

Odpočítavanie

```
n = 5
while n > 0:
    print(n)
    n -= 1

print("Štart!")
```

Čakanie na správny vstup

```
heslo = ""
while heslo != "tajne123":
    heslo = input("Zadaj heslo: ")
print("Vitaj!")
```

Nekonečný cyklus (len ak ho potrebuješ)

```
while True:
    prikaz = input("Napíš príkaz (q = koniec): ")
    if prikaz == "q":
        break
```

Všimnime si slovo "break" na konci programu. Toto slovo zabezpečí, že celý program prestane bežať. Pokiaľ teda príde z konzoly od používateľa prikaz "q", potom program prečíta break a skončí.

Pozor na chyby

Zabudnutá zmena premennej

```
i = 1
while i <= 5:
    print(i)
    # i sa nikdy nezmení → nekonečný cyklus
```

Musíš meniť hodnotu vnútri cyklu. Napríklad $i += 1$.

Podmienka, ktorá nikdy nie je False

```
x = 10
while x > 0:
    print(x)
    # x sa neznižuje → nekonečný cyklus
```

Mini cvičenia

1. Vypíš čísla od 1 do 10 pomocou while.
2. Počítaj od 10 dole po 1 (odpočítavanie).
3. Požiadaj používateľa, aby zadal číslo väčšie ako 100 → opakuj, kým to nesprávne.
4. Urob program, ktorý sa pýta na heslo, až kým nezadá správne.

Vnorené cykly

Čo je vnorený cyklus?

Vnorený cyklus = cyklus vo vnútri iného cyklu.

Používa sa v situáciách, kde treba prechádzať **dve dimenzie** alebo kombinácie hodnôt:

- tabuľka násobilky,
- dvojrozmerné zoznamy (matice),
- tvorenie obrazcov,
- hry (herná mapa, hracie pole).

Základná štruktúra

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

Výstup:

```
0 0
0 1
1 0
1 1
2 0
2 1
```

Tabuľka násobilky

```
for i in range(1, 6):
    for j in range(1, 6):
        print(f"{i*j:3}", end=" ")
    print()
```

Výstup:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

Hviezdičkové obrazce

Pravouhlý trojuholník

```
for i in range(1, 6):
    for j in range(i):
        print("*", end="")
    print()
```

Výstup:

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

Šachovnica

```
for i in range(5):
    for j in range(5):
        if (i + j) % 2 == 0:
            print("X", end=" ")
        else:
            print("0", end=" ")
    print()
```

Typické chyby

Príliš veľa úrovní vnorenia → chaos.

Riešenie: rozdeliť kód do funkcií.

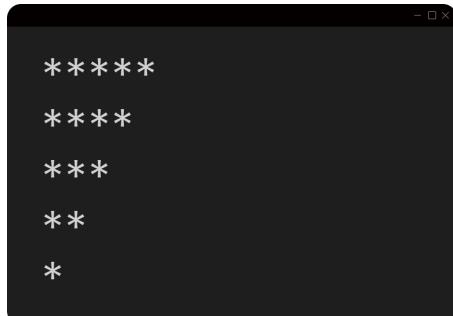
Zabudnutý `print()` na konci riadku.

Výsledok sa „zleje“ do jedného riadku.

Zlá logika v podmienkach → obrazec nevychádza.

Mini cvičenia

1. Vypíš násobilku pre čísla 1–10 vo forme tabuľky.
2. Vytvor obrazec:



1. Skús spraviť šachovnicu 8×8 s písmenami „B“ a „Č“.
2. Prejdi 2D zoznam známok a vypíš priemery pre každý riadok.

Sekcia V: Dátové štruktúry

Listy (zoznamy)

Prečo potrebujeme zoznamy (listy)?

Predstav si, že chceš uložiť 100 známok.

Bez zoznamu by si musel mať 100 premenných (znamka1, znamka2 ...).

V Pythone máme **listy** = premenná, ktorá dokáže obsahovať viac hodnôt naraz.

Vytvorenie zoznamu

```
ovocie = ["jablko", "banán", "hruška"]
cisla = [1, 2, 3, 4, 5]
mix = [1, "Python", 3.14, True]    # rôzne typy naraz
```

Prázdny zoznam:

```
prazdny = []
```

Prístup k prvkom

```
znamky = [1, 2, 3, 1, 2]

print(znamky[0])    # 1 (prvý prvok)
print(znamky[-1])  # 2 (posledný prvok)
```

Všimnite si, že *indexovanie* je úplne rovnaké, ako pri reťazcoch.

Zmena hodnoty

```
znamky[1] = 5
print(znamky)    # [1, 5, 3, 1, 2]
```

Užitočné metódy listov

- `append(x)` → pridá prvok na koniec
- `insert(i, x)` → vloží na pozíciu
- `remove(x)` → vymaže prvý výskyt
- `pop(i)` → vymaže na pozícii a vráti ho
- `sort()` → usporiada
- `reverse()` → otočí
- `count(x)` → spočíta výskyty
- `index(x)` → pozícia prvku

Príklady:

```
- □ ×  
znamky = [1, 2, 3, 1, 2]  
  
znamky.append(4)      # [1, 2, 3, 1, 2, 4]  
znamky.remove(1)      # vymaže prvú 1 → [2, 3, 1, 2, 4]  
znamky.sort()         # [1, 2, 2, 3, 4]  
print(znamky.count(2)) # 2
```

Prechádzanie listu cyklom

```
- □ ×  
ovocie = ["jablko", "banán", "hruška"]  
  
for item in ovocie:  
    print(item)
```

S indexami:

```
- □ ×  
for i, item in enumerate(ovocie):  
    print(i, item)
```

Typické chyby

Index mimo rozsahu

```
cisla = [10, 20, 30]
print(cisla[5])    # IndexError
```

Zámena `remove()` a `pop()`

- `remove(hodnota)` → odstráni prvú hodnotu, ak existuje.
- `pop(index)` → odstráni na pozícii.

Mini cvičenia

1. Vytvor zoznam svojich 5 obľúbených jedál a vypíš ich.
2. Uprav druhé jedlo na „pizza“.
3. Pridaj do zoznamu nové jedlo pomocou `append()`.
4. Spočítaj, kol'kokrát sa v zozname vyskytuje „pizza“.
5. Vytvor zoznam čísel `[5, 3, 8, 1]`, usporiadaj ho a vypíš najväčšie číslo.

Sety (množiny)

Čo je set (množina)?

- **Neusporiadaná kolekcia** unikátnych prvkov.
- Automaticky vyhadzuje duplicity.
- Nemá indexy (nemôžeš volať `set[0]`).

Príklad:

```
cisla = {1, 2, 3, 3, 4}  
print(cisla)    # {1, 2, 3, 4}
```

Vytvorenie setu

```
prazdny = set()  
ovocie = {"jablko", "banán", "hruška"}
```

Pozor: {} samotné vytvorí **prázdny slovník**, nie set.

Pridávanie a mazanie prvkov

```
ovocie = {"jablko", "banán"}  
  
ovocie.add("hruška")  
print(ovocie)    # {'jablko', 'banán', 'hruška'}  
  
ovocie.remove("banán")  
print(ovocie)    # {'jablko', 'hruška'}  
  
ovocie.discard("slivka") # nevyhodí chybu, ak prvok neexistuje
```

Množinové operácie

Sety sú super na porovnávanie a kombinovanie dát.

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A | B)      # zjednotenie {1, 2, 3, 4, 5}
print(A & B)      # prienik {3}
print(A - B)      # rozdiel {1, 2}
print(A ^ B)      # symetrický rozdiel {1, 2, 4, 5}
```

Užitočné metódy

- `len(set)` → počet prvkov
- `in` → kontrola, či prvak existuje
- `set1.issubset(set2)` → podmnožina
- `set1.issuperset(set2)` → nadmnožina

Typické chyby

Snažiť sa pristupovať k prvku cez index

```
s = {1, 2, 3}
print(s[0])    # TypeError
```

SPRÁVNE: iterovať

```
for x in s:
    print(x)
```

Očakávať poradie prvkov → set je neusporiadaný.

Mini cvičenia

1. Vytvor set mien spolužiakov a pridaj tam svoje meno.
2. Zo zoznamu [1, 2, 2, 3, 4, 4, 5] vytvor set a zisti, koľko unikátnych čísel obsahuje.
3. Zisti, ktoré písmená sa nachádzajú v oboch slovách "python" a "java".
4. Vytvor dva sety čísel a zisti ich prienik aj zjednotenie.

Tuple (n-tice)

1) Čo je tuple?

- **Nemenný zoznam** → hodnoty v ňom sa **nedajú meniť** po vytvorení.
- Používa sa, keď chceš dátá uložiť tak, aby sa „nepokazili“.
- Rýchlejšie a úspornejšie než listy.

Vytvorenie tuple

```
prazdny = ()  
farby = ("červená", "zelená", "modrá")  
cisla = (1, 2, 3, 4)
```

Jeden prvok v tuple:

```
jedno = (5,)    # musí mať čiarku
```

Prístup k prvkom

```
farby = ("červená", "zelená", "modrá")  
  
print(farby[0])    # červená  
print(farby[-1])  # modrá
```

Nemennosť (immutability)

```
farby = ("červená", "zelená", "modrá")  
farby[0] = "žltá"  # TypeError - tuple sa nedá meniť
```

Ak potrebujete zmenu → premeň tuple na list:

```
zoznam = list(farby)  
zoznam[0] = "žltá"  
print(zoznam)
```

Rozbalenie tuple (unpacking)

```
osoba = ("Jana", 17)
meno, vek = osoba
print(meno) # Jana
print(vek) # 17
```

Užitočné operácie

- `len(tuple)` → počet prvkov
- `in` → kontrola výskytu
- `count(x)` → počet výskytov
- `index(x)` → pozícia prvku

```
cisla = (1, 2, 2, 3)
print(cisla.count(2)) # 2
print(cisla.index(3)) # 3
```

Typické chyby

Zabudnutá čiarka pri jednom prvku

```
x = (5)      # nie je tuple, ale int
y = (5,)     # tuple
```

Pokus o zmenu prvku → `TypeError`.

Mini cvičenia

1. Vytvor tuple so svojím menom a vekom. Rozbal' ho na dve premenné.
2. Urob tuple s 5 číslami, vypíš ich súčet a priemer.
3. Skús nájsť index čísla 7 v tuple `(3, 5, 7, 9)`.
4. Skús spraviť tuple s jedným prvkom – čo sa stane, keď vynecháš čiarku?

Slovníky (dictionaries)

Čo je slovník?

- Kolekcia dát vo forme **kľúč** → **hodnota**.
- Namiesto indexov (0,1,2...) používame **kľúče** (napr. reťazce).
- Rýchly prístup k údajom – ak poznáš kľúč, vieš sa dostať k hodnote.

Príklad:

```
student = {  
    "meno": "Jana",  
    "vek": 17,  
    "trieda": "3.B"  
}
```

Všimnite si, kľúče sú vždy hodnoty/názvy hodnôt vľavo a k nim sa priraduje daná hodnota (text, číslo...).

Prístup k hodnotám

```
print(student["meno"]) # Jana  
print(student["vek"]) # 17
```

Bezpečnejší prístup cez `.get()` → vráti `None`, ak kľúč neexistuje.

```
print(student.get("adresa")) # None
```

Pridanie a zmena hodnôt

```
student["vek"] = 18 # zmena  
student["mesto"] = "Bratislava" # pridanie  
print(student)
```

Mazanie hodnôt

```
- □ ×  
student.pop("trieda")      # vymaže klúč "trieda"  
del student["vek"]        # vymaže klúč "vek"
```

Užitočné metódy

```
- □ ×  
print(student.keys())    # všetky klúče  
print(student.values())  # všetky hodnoty  
print(student.items())   # klúče aj hodnoty
```

Iterovanie cez slovník:

```
- □ ×  
for k, v in student.items():  
    print(k, "->", v)
```

Vnorené slovníky

Slovník môže obsahovať ďalší slovník (alebo inú dátovú štruktúru).

```
- □ ×  
trieda = {  
    "student1": {"meno": "Jana", "vek": 17},  
    "student2": {"meno": "Peter", "vek": 18}  
}  
print(trieda["student1"]["meno"])  # Jana
```

Typické chyby

Pokus o prístup k neexistujúcemu klúču

```
- □ ×  
print(student["adresa"])  # KeyError
```

Riešenie: `get()` alebo podmienka `if "adresa" in student.`

Myslieť si, že slovník má poradie → až od Python 3.7 sa uchováva poradie vloženia, ale logika na tom nesmie závisieť.

Mini cvičenia

1. Vytvor slovník so svojím menom, vekom a mestom. Vypíš ho.
2. Zmeň hodnotu „vek“ na +1.
3. Pridaj novú položku „škola“.
4. Vypíš všetky kľúče slovníka.
5. Vytvor slovník triedy s aspoň 3 študentmi (vnorený slovník).

Sekcia VI: Náhodnosť a interaktivita v programovaní

Random

Prečo potrebujeme náhodnosť?

Náhodnosť sa používa všade:

- hádzanie kockou, miešanie kariet, generovanie hesiel,
- náhodný výber otázok v teste,
- simulácie (počasie, štatistiky),
- hry (random spawn, súboje, AI správanie).

V Pythone pracujeme s náhodou cez modul `random`.

Import modulu

```
import random
```

Najčastejšie funkcie

`random.random()` – náhodné desatinné číslo 0–1

```
print(random.random()) # napr. 0.7348
```

`random.randint(a, b)` – celé číslo od a do b (vrátane)

```
print(random.randint(1, 6)) # simulácia kocky
```

`random.uniform(a, b)` – desatinné číslo v rozsahu

```
print(random.uniform(1.5, 5.5))
```

`random.choice(sekvencia)` – náhodný prvok zo zoznamu alebo stringu

```
farby = ["červená", "zelená", "modrá"]
print(random.choice(farby))
```

random.shuffle(zoznam) – premiešanie zoznamu

```
karty = ["A", "K", "Q", "J"]
random.shuffle(karty)
print(karty)
```

random.sample(zoznam, n) – náhodný výber n unikátnych prvkov

```
cisla = [1, 2, 3, 4, 5, 6]
print(random.sample(cisla, 3))
```

Typické chyby

Zabudnutý import random.

Používanie randint() mimo rozsahu (napr. randint(1, 0)).

Zámena choice() a sample():

- choice() → jeden prvek,
- sample() → viac prvkov bez opakovania.

Mini cvičenia

1. Napíš program, ktorý hodí kockou (čísla 1–6).
2. Vygeneruj 10 náhodných čísel od 0 do 100.
3. Náhodne vyber meno zo zoznamu ["Eva", "Peter", "Marek"].
4. Vytvor „náhodné heslo“ z 8 písmen (použi choice() zo stringu).
5. Vytvor program, ktorý premieša zoznam kariet ["♥A", "♦K", "♣Q", "♠J"].

Sekcia VII: Tvorba funkcií a správa parametrov

Funkcie

Prečo vlastne potrebujeme funkcie?

Predstav si, že máš napísť program, ktorý počíta obvod obdĺžnika.

Napíšeš:

```
a = 5  
b = 3  
print(2 * (a + b))
```

Super. A teraz si predstav, že potrebuješ vypočítať obvod pre 20 rôznych obdĺžnikov.

Čo spravíš? Nakopíruješ tento kód 20-krát a vždy vymeníš hodnoty **a** a **b**?

To je **strašne neefektívne** a robí to bordel v kóde.

Tu prichádzajú na scénu **funkcie**.

Funkcia je ako **krabička na kód** – raz ju vytvoríš a potom ju môžeš používať stále dokola, vždy keď potrebuješ.

Ako vyzerá funkcia v Pythone?

Základná štruktúra funkcie je:

```
def nazov_funkcie():  
    # tu je kód, ktorý sa vykoná
```

- **def** = kľúčové slovo, ktoré hovorí: „idem definovať funkciu“
- **nazov_funkcie** = meno, ktoré si zvolíš (napr. **pozdrav**, **vypocitaj_obvod**)
- **()** = zátvorky, do ktorých môžeme dať parametre (o chvíľu si povieme viac)
- **:** = dvojbodka (nezabudni na ňu, bez nej to nejde)
- pod tým → riadky kódu, ktoré patria do funkcie (odsadené 4 medzerami alebo tabom)

Príklad:

```
def pozdrav():  
    print("Ahoj, vitaj v Pythone!")
```

A teraz ju spustíme:

```
pozdrav()
```

Výstup:

```
Ahoj, vitaj v Pythone!
```

Vidíš? Funkcia je kód, ktorý si môžeš spustiť na požiadanie.

Parametre – čo funkcia „požiera“

Funkcia je užitočná hlavne vtedy, keď jej **posielame hodnoty**, s ktorými pracuje. Tieto hodnoty voláme **parametre**.

```
def pozdrav(meno):  
    print(f"Ahoj {meno}!")
```

Ked' teraz zavoláme funkciu:

```
pozdrav("Nina")  
pozdrav("Peter")
```

Výstup:

```
Ahoj Nina!  
Ahoj Peter!
```

Všimni si, že namiesto toho, aby sme písali dvakrát podobný kód, stačí funkciu len zavolať s iným parametrom.

Viac parametrov

Funkcia môže mať aj viac parametrov:

```
def sucet(a, b):  
    print(a + b)  
  
sucet(3, 5) # 8  
sucet(10, 20) # 30
```

Toto je užitočné, keď chceme prepočítavať rôzne vstupy stále rovnakým spôsobom.

Návratová hodnota (return)

Nie vždy chceme, aby funkcia **len niečo vypísala**.

Niekedy chceme, aby funkcia **vrátila výsledok**, s ktorým môžeme ďalej pracovať.

Na to používame kľúčové slovo **return**.

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

vysledok = sucet(7, 4)
print(vysledok)      # 11
print(vysledok * 2)  # 22
```

Rozdiel:

- **print()** iba zobrazí na obrazovku.
 - **return** vracia hodnotu, ktorú vieme uložiť a ďalej používať.
-

Predvolené hodnoty parametrov

Môžeme nastaviť, aby mal parameter **predvolenú hodnotu**, ak ho nezadáme.

```
def pozdrav(meno="Študent"):
    print(f"Ahoj {meno}!")

pozdrav()      # Ahoj Študent!
pozdrav("Marek")  # Ahoj Marek!
```

Typické chyby

Zabudnuté zátvorky pri volaní funkcie:

```
pozdrav      # nič sa nestane
```

Správne: **pozdrav()**

Zámena **print()** a **return**:

```
def sucet(a, b):
    print(a + b)

x = sucet(2, 3)
print(x)      # vypíše 5 a potom None
```

Správne:

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

Rovnaký názov funkcie a premennej:

```
sum = 5
def sum(a, b):  # toto Pythonu prekáža
    return a+b
```

Mini cvičenia

1. Napíš funkciu `obvod_obdlznika(a, b)`, ktorá vráti obvod obdĺžnika.
2. Napíš funkciu `plocha_kruhu(r)`, ktorá vráti obsah kruhu (πr^2).
3. Napíš funkciu `maximum(a, b)`, ktorá vráti väčšie číslo.
4. Napíš funkciu `is_even(x)`, ktorá vráti `True`, ak je číslo párné.

Argumenty, *args, **kwargs

Zopakujme si: parametre vs. argumenty

- **Parametre** = mená vo vnútri definície funkcie.
- **Argumenty** = konkrétné hodnoty, ktoré funkciu posielame.

```
def pozdrav(meno):    # meno = parameter
    print(f"Ahoj {meno}!")

pozdrav("Nina")      # "Nina" = argument
```

Niekedy však **nevieme dopredu**, kolko argumentov bude mať funkcia. A presne preto existujú ***args** a ****kwargs**.

*args – neobmedzený počet pozičných argumentov

***args** umožní poslať funkciu ľubovoľný počet argumentov → vo vnútri sa spracujú ako **tuple**.

```
def sucet(*args):
    vysledok = 0
    for cislo in args:
        vysledok += cislo
    return vysledok

print(sucet(1, 2, 3))      # 6
print(sucet(10, 20, 30, 40))# 100
```

Funkcia je univerzálna: nemusíš dopredu vedieť, kolko čísel budeš sčítovať.

**kwargs – neobmedzený počet pomenovaných argumentov

****kwargs** spracuje argumenty vo forme **kľúč = hodnota** → vo vnútri sa uložia ako **slovník**.

```
def profil(**kwargs):
    for kluc, hodnota in kwargs.items():
        print(f"{kluc}: {hodnota}")

profil(meno="Nina", vek=17, mesto="Bratislava")
```

Výstup:

```
meno: Nina  
vek: 17  
mesto: Bratislava
```

Toto je výborné, keď chceme flexibilne ukladať informácie bez pevne určeného počtu parametrov.

Kombinovanie parametrov

Môžeme použiť ****klasické parametre + *args + kwargs** v jednej funkcií.

```
def objednavka(id, *polzky, **detaily):  
    print(f"ID objednávky: {id}")  
    print("Položky:", polzky)  
    print("Detaily:", detaily)  
  
objednavka(  
    123,  
    "pizza", "cola", "salat",  
    meno="Peter",  
    adresa="Hlavná 5"  
)
```

Výstup:

```
ID objednávky: 123  
Položky: ('pizza', 'cola', 'salat')  
Detaily: {'meno': 'Peter', 'adresa': 'Hlavná 5'}
```

Poradie argumentov

Pri definovaní funkcie platí poradie:

1. normálne parametre
2. ***args**
3. ****kwargs**

```
def funkcia(a, b, *args, **kwargs):  
    pass
```

Typické chyby

Zabudnutá hviezdička → funkcia čaká iba jeden parameter.

```
def test(args):  
    print(args)  
  
test(1, 2, 3)    # TypeError
```

Správne:

```
def test(*args):  
    print(args)
```

Snažiť sa indexovať `**kwargs` ako zoznam.

```
def test(**kwargs):  
    print(kwargs[0])    # KeyError
```

Správne: pracujeme cez kľúče.

Mini cvičenia

1. Napíš funkciu `sucet(*args)`, ktorá spočíta všetky čísla.
2. Napíš funkciu `informacie(**kwargs)`, ktorá vypíše meno, vek a školu (hodnoty zadáš pri volaní).
3. Skombinuj: vytvor funkciu `objednavka(*args, **kwargs)`, ktorá vypíše položky objednávky a meno zákazníka.
4. Skús vytvoriť funkciu, ktorá vráti najväčšie číslo zo všetkých argumentov (`max(*args)`).

Iterables (iterovateľné objekty)

Čo je iterable?

- **Iterable = objekt, cez ktorý vieme iterovať** (prechádzať prvok po prvku).
- Medzi základné iterables patria: **stringy, listy, tuple, sety, slovníky, range()**.
- Každý iterable vieme prechádzať pomocou **for cyklu**.

Príklad:

```
for znak in "Python":  
    print(znak)
```

for cyklus nad iterables

```
cisla = [10, 20, 30]  
for c in cisla:  
    print(c)
```

Všimni si, že nemusíme riešiť indexy – Python sa sám stará o to, aby nám dával prvky po jednom.

Funkcia **map()** – aplikovanie funkcie na každý prvok

```
def na_druhu(x):  
    return x ** 2  
  
cisla = [1, 2, 3, 4]  
druhe = list(map(na_druhu, cisla))  
print(druhe) # [1, 4, 9, 16]
```

map() zoberie funkciu + iterable → vráti nový iterable s upravenými prvkami.

Krátšia verzia s **lambda**:

```
cisla = [1, 2, 3, 4]  
print(list(map(lambda x: x**2, cisla)))
```

Funkcia **filter()** – vyberanie prvkov podľa podmienky

```
def je_porne(x):
    return x % 2 == 0

cisla = [1, 2, 3, 4, 5, 6]
parne = list(filter(je_porne, cisla))
print(parne) # [2, 4, 6]
```

`filter()` prechádza cez iterable a nechá iba tie prvky, pre ktoré funkcia vráti `True`.

Generátory a comprehension (mini ochutnávka)

```
cisla = [1, 2, 3, 4, 5]
druhe = [x**2 for x in cisla]
print(druhe) # [1, 4, 9, 16, 25]
```

Toto je vlastne kombinácia `for` + `map()` v krátkom zápise.

Iterácia cez slovníky

```
student = {"meno": "Eva", "vek": 17}

for kluc, hodnota in student.items():
    print(kluc, "->", hodnota)
```

Typické chyby

Zabudnutie pretypovať `map()` alebo `filter()` na list

```
cisla = [1, 2, 3]
print(map(lambda x: x*2, cisla)) # <map object>
```

Správne: `list(map(...))`

Snažiť sa indexovať set → `TypeError`.

Myslieť si, že `filter()` zmení pôvodný zoznam – nie, vráti nový iterable.

Mini cvičenia

1. Použi `map()`, aby si premenil zoznam mien `["eva", "peter", "jana"]` na veľké písmená.
2. Použi `filter()`, aby si zistil, ktoré čísla zo zoznamu `[3, 7, 8, 12, 15]` sú väčšie než 10.

3. Použi comprehension, aby si zo zoznamu [1, 2, 3, 4, 5] urobil zoznam kociek.
4. Prejdi cez string "RoboSkillz" a vypíš všetky znaky oddelené medzerou.

Sekcia VIII: Efektívna syntax

List comprehensions

Prečo list comprehensions?

Doteraz sme robili nové zoznamy cez cyklus:

```
cisla = [1, 2, 3, 4, 5]
druhe = []
for x in cisla:
    druhe.append(x**2)
print(druhe)  # [1, 4, 9, 16, 25]
```

Funguje to, ale je to zdĺhavé.

List comprehension je elegantnejší zápis:

```
cisla = [1, 2, 3, 4, 5]
druhe = [x**2 for x in cisla]
print(druhe)  # [1, 4, 9, 16, 25]
```

Základná štruktúra

```
novy_list = [výraz for prvok in iterable]
```

- **iterable** = kolekcia, cez ktorú prechádzame (napr. list, range, string).
 - **prvok** = aktuálna hodnota.
 - **výraz** = čo sa s prvkom spraví.
-

Jednoduché príklady

Mocniny čísel

```
mocniny = [x**2 for x in range(1, 6)]
print(mocniny)  # [1, 4, 9, 16, 25]
```

Veľké písmená

```
mena = ["eva", "peter", "jana"]
mena_upravene = [m.capitalize() for m in mena]
print(mena_upravene)  # ['Eva', 'Peter', 'Jana']
```

S podmienkou (if)

Do comprehension môžeme pridať filter.

```
cisla = [1, 2, 3, 4, 5, 6, 7, 8, 9]
parne = [x for x in cisla if x % 2 == 0]
print(parne) # [2, 4, 6, 8]
```

S if-else vnútri výrazu

```
hodnoty = [x if x >= 0 else 0 for x in [-3, -1, 2, 4]]
print(hodnoty) # [0, 0, 2, 4]
```

Vnorené comprehension

Môžeme prechádzať viac iterables naraz.

```
kombinacie = [(x, y) for x in [1, 2, 3] for y in [4, 5, 6]]
print(kombinacie)
# [(1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6)]
```

Typické chyby

Prehnané vnáranie comprehension → kód sa stane nečitateľným.

Riešenie: použiť radšej klasický cyklus.

Zabudnuté hranaté zátvorky → vytvorí sa generator, nie list.

```
(x**2 for x in range(5)) # generator
[x**2 for x in range(5)] # list
```

Používanie comprehension tam, kde nepotrebuješ nový list → zbytočná spotreba pamäte.

Mini cvičenia

1. Vytvor zoznam obsahujúci kocky čísel 1–10.
2. Zo zoznamu [3, 7, 8, 12, 15] vytvor nový zoznam len s číslami väčšími ako 10.
3. Zo zoznamu mien ["anna", "michal", "eva"] sprav zoznam s veľkými písmenami.
4. Vytvor zoznam dvojíc (x, y) pre všetky kombinácie čísel od 1 do 3 a od 1 do 2.
5. Použi comprehension s if-else, ktoré všetky záporné čísla nahradí nulou.

Match-case

Prečo match-case?

Doteraz sme robili rozhodovanie pomocou if/elif/else.

Napríklad:

```
znamka = 2

if znamka == 1:
    print("Výborný")
elif znamka == 2:
    print("Chválitebný")
elif znamka == 3:
    print("Dobrý")
elif znamka == 4:
    print("Dostatočný")
else:
    print("Nedostatočný")
```

Funguje to, ale pri veľkom počte možností je to **neprehľadné**.

Python 3.10 priniesol match-case, ktoré robí kód čitateľnejším.

Základná štruktúra

```
match hodnota:
    case moznost1:
        # kód
    case moznost2:
        # kód
    case _:
        # iné (default)
```

Jednoduchý príklad – známky

```
znamka = 2

match znamka:
    case 1:
        print("Výborný")
    case 2:
        print("Chválitebný")
    case 3:
        print("Dobrý")
    case 4:
        print("Dostatočný")
    case _:
        print("Nedostatočný")
```

Viac hodnôt v jednom case

```
ovocie = "jablko"

match ovocie:
    case "jablko" | "hruška":
        print("Ovocie")
    case "mrkva" | "zeler":
        print("Zelenina")
    case _:
        print("Neznáme")
```

Match s podmienkou (if guard)

```
cislo = 12

match cislo:
    case x if x % 2 == 0:
        print("Párne číslo")
    case x if x % 2 != 0:
        print("Nepárne číslo")
```

Rozbalenie (pattern matching)

```
bod = (3, 4)

match bod:
    case (0, 0):
        print("Začiatok")
    case (x, 0):
        print(f"Bod na osi X: {x}")
    case (0, y):
        print(f"Bod na osi Y: {y}")
    case (x, y):
        print(f"Súradnice: {x}, {y}")
```

Toto je obrovská sila **match** – dokáže priamo rozbalíť štruktúru a priradiť premenné.

Typické chyby

Zabudnutý **_** ako „default“ case → program nemusí pokryť všetky možnosti.

Používanie v staršej verzii Pythonu (< 3.10) → **SyntaxError**.

Prehnанé používanie → ak je len 2–3 možnosti, if/else je jednoduchší.

Mini cvičenia

1. Použi **match-case**, aby program vypísal deň v týždni podľa čísla (1 = pondelok, ... 7 = nedel'a).
2. Napíš program, ktorý podľa mena predmetu (napr. „matematika“, „dejepis“) vypíše, či patrí medzi prírodovedné alebo humanitné.
3. Skús pomocou **match-case** rozpoznať súradnice bodu (x, y) – či je v strede, na osi X, osi Y alebo inde.

Moduly

Čo je modul?

- **Modul = súbor s Python kódom**, ktorý môžeme použiť v iných programoch.
- V module môžu byť **funkcie, triedy, premenné**.
- V Pythone existujú:
 - vstavané moduly (už sú súčasťou Pythonu, napr. `math`, `random`),
 - vlastné moduly (napísané nami),
 - externé moduly (inštalované cez `pip`, napr. `numpy`, `requests`).

Moduly nám šetria čas – namiesto toho, aby sme všetko písali sami, použijeme existujúce riešenia.

Použitie vstavaných modulov

```
import math  
  
print(math.sqrt(16))      # 4.0  
print(math.pi)           # 3.14159...
```

`import` = prikádzanie Pythonu, aby načítal modul.

Alias (skrátený názov)

```
import math as m  
print(m.sqrt(25))
```

Len konkrétné funkcie

```
from math import sqrt, pi  
print(sqrt(9))  
print(pi)
```

Vlastný modul

1. Vytvor súbor `moj_modul.py`:

```
def pozdrav(meno):  
    return f"Ahoj {meno}!"
```

1. V inom súbore:

```
import moj_modul  
print(moj_modul.pozdrav("Nina"))
```

Takto si môžeme rozdeliť väčší projekt na viacero súborov.

Externé moduly (cez pip)

Python má obrovský ekosystém.

Inštalácia balíka:

```
pip install requests
```

Použitie:

```
import requests  
  
response = requests.get("https://api.github.com")  
print(response.status_code)
```

Typické chyby

Zabudnutý import → NameError.

Konflikt názvov – vlastný súbor sa volá rovnako ako vstavaný modul (napr. random.py).

Zabudnutý pip install pri externých knižniciach.

Mini cvičenia

1. Importuj math a vypočítaj obvod kruhu so zadaným polomerom.
2. Použi random.randint(1, 10) na hádzanie kockou.
3. Vytvor vlastný modul utils.py, kde bude funkcia plocha_obdlnnika(a, b). Importuj ju do iného súboru.
4. Skús nainštalovať balík emoji (pip install emoji) a vypíš jednoduchý text s emoji.

Scope (lokálny a globálny)

Čo je scope?

Scope = **oblasť platnosti premennej** → teda kde v programe premenná existuje a kde k nej máme prístup.

V Pythone sú najčastejšie:

- **Lokálne premenné** – platia iba vo vnútri funkcie.
- **Globálne premenné** – platia v celom súbore (module).

Lokálna premenná

```
def moja_funkcia():
    x = 10
    print("Vo funkcií:", x)

moja_funkcia()
print("Mimo funkcie:", x) # chyba!
```

Výstup:

```
- □ ×
Vo funkcií: 10
NameError: name 'x' is not defined
```

Premenná **x** existuje len vo vnútri funkcie → mimo nej ju Python nepozná.

Globálna premenná

```
- □ ×
x = 100    # globálna premenná

def vypis():
    print("Vo funkcií:", x)

vypis()
print("Mimo funkcie:", x)
```

Výstup:

```
Vo funkcii: 100  
Mimo funkcie: 100
```

Globálne premenné sú dostupné vo funkciách, **ale len na čítanie** (pokiaľ ich priamo nemeníme).

Zmena globálnej premennej vo funkcií

Ak chceme zmeniť globálnu premennú, musíme použiť kľúčové slovo **global**.

```
pocitadlo = 0  
  
def pridaj():  
    global pocitadlo  
    pocitadlo += 1  
  
pridaj()  
print(pocitadlo) # 1
```

Pozor – používanie **global** sa vo veľkých projektoch neodporúča (ľahko vznikne chaos). Radšej používame **parametre a návratové hodnoty**.

Shadowing – keď lokálna premenná „prebije“ globálnu

```
x = 50  
  
def test():  
    x = 10  
    print("Vo funkcii:", x)  
  
test()  
print("Mimo funkcie:", x)
```

Výstup:

```
Vo funkcii: 10  
Mimo funkcie: 50
```

Premenná **x** vo vnútri funkcie **zatiaľ** globálnu **x**.

Scope vnútri funkcií (LEGB pravidlo)

Python hľadá premennú v poradí:

1. **Local** – vo vnútri aktuálnej funkcie,
2. **Enclosing** – vo vnorených funkciách,
3. **Global** – na úrovni modulu,
4. **Built-in** – vstavané názvy ako `print`, `len`.

Toto pravidlo sa volá **LEGB**.

Typické chyby

Pokus používať premennú, ktorá je len lokálna:

```
def funkcia():
    y = 5
    print(y)    # NameError
```

Neúmyselné prepísanie globálnej premennej:

```
x = 100
def funkcia():
    x = 5    # študent si myslí, že mení globálne x, ale mení lokálne
```

Mini cvičenia

1. Vytvor funkciu, ktorá vo vnútri používa lokálnu premennú `x`. Skús ju vypísať mimo funkcie – čo sa stane?
2. Vytvor globálnu premennú `skore = 0`. Potom funkciu, ktorá ju pomocou `global` zvyšuje.
3. Vyskúšaj „shadowing“ – urob globálne `x = 100` a lokálne `x = 20`. Vypíš obe hodnoty.
4. Vytvor vnorenú funkciu (funkcia vo funkcií), ktorá použije premennú z „enclosing scope“.

Sekcia IX: Štruktúrovanie kódu a práca s hlavnými skriptmi

Main

Problém bez `_main_`

Predstav si, že máš súbor `matika.py`:

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

print(sucet(2, 3))
```

Ak tento súbor spustíš, vypíše:

```
5
```

Ale čo sa stane, keď ho importuješ do iného programu?

```
import matika
```

Výstup bude:

```
5
```

Nechceli sme len importovať funkciu, ale zároveň sa vykonal aj `print`!

Prečo sa to deje?

Ked' Python spustí súbor, automaticky nastaví špeciálnu premennú `_name_`.

- Ak súbor spúštaš priamo → `_name_ = "__main__"`.
- Ak súbor importuješ → `_name_ = "názov_modulu"`.

Ako to vyriešiť?

Použijeme podmienku:

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

if __name__ == "__main__":
    print(sucet(2, 3))
```

Teraz:

- Ak spustíš súbor priamo → vykoná sa `print(sucet(2, 3))`.
- Ak importuješ súbor → nič sa nevypíše, len máš k dispozícii funkciu `sucet()`.

Štruktúra profesionálnych programov

Bežný Python projekt:

```
# matika.py
def sucet(a, b):
    return a + b

def rozdiel(a, b):
    return a - b

if __name__ == "__main__":
    print("Testovanie modulu matika")
    print(sucet(5, 3))
    print(rozdiel(5, 3))
```

Takto modul obsahuje:

- **definície funkcií** → na import,
- **blok if __name__ == "__main__"** → na testovanie a spúšťanie ako hlavný program.

Typické chyby

Zabudnutie na `__name__ == "__main__"` → modul sa správa inak pri importe, než by si čakal.

Písanie logiky programu mimo funkcií a hlavného bloku → chaos, kód sa spúšťa nekontrolované.
Nepochopenie rozdielu medzi modulom a skriptom → študenti často nechápu, prečo sa kód „sám spustil“.

Mini cvičenia

1. Vytvor súbor `pozdrav.py` s funkciou `pozdrav(meno)` a blokom `if __name__ == "__main__"`.
 - Spusti ho priamo.
 - Potom ho importuj do iného súboru a otestuj rozdiel.
2. Uprav svoj starší program „kalkulačka“ tak, aby mal funkcie (`scitaj`, `odcitaj`, ...) a testovací blok v `__main__`.
3. Skús spraviť modul `geometria.py` s funkciami `obvod_kruhu(r)` a `plocha_kruhu(r)`. Otestuj ich cez `__main__`.

Sekcia X: Objektovo orientované programovanie

Základy objektovo orientovaného programovania (OOP)

Prečo OOP?

Doteraz sme robili programy ako sériu príkazov a funkcií. To je v pohode pri malých projektoch. Ale čo ak chceme modelovať **reálne objekty** → auto, účet v banke, žiaka v škole?

- Funkcie samy osebe nestačia – museli by sme mať veľa premenných (meno, vek, trieda, známky).
- V OOP vieme tieto dátá a funkcie spojiť do **jedného balíka = trieda**.

OOP = programovanie pomocou **objektov**.

- **Trieda** = šablóna (plán domu).
- **Objekt** = konkrétna inštancia triedy (postavený dom).

Základná štruktúra triedy

```
class Student:  
    def __init__(self, meno, vek):  
        self.meno = meno  
        self.vek = vek  
  
    def pozdrav(self):  
        print(f"Ahoj, volám sa {self.meno} a mám {self.vek} rokov.")
```

Vysvetlenie:

- `class Student:` → definujeme triedu.
- `__init__` → konštruktor, ktorý sa spustí pri vytvorení objektu.
- `self` → odkaz na konkrétnu inštanciu (objekt).
- `self.meno, self.vek` → vlastnosti (atribúty).
- `pozdrav()` → metóda (funkcia vo vnútri triedy).

Vytváranie objektov (inštancií)

```
s1 = Student("Eva", 17)  
s2 = Student("Peter", 18)  
  
s1.pozdrav()  
s2.pozdrav()
```

Výstup:

```
Ahoj, volám sa Eva a mám 17 rokov.  
Ahoj, volám sa Peter a mám 18 rokov.
```

Každý objekt má svoje **vlastné dátá**.

Premenné triedy vs. inštančné premenné

```
class Auto:  
    kolesa = 4    # premenná triedy (rovnaká pre všetky autá)  
  
    def __init__(self, znacka):  
        self.znacka = znacka # inštančná premenná  
  
a1 = Auto("Tesla")  
a2 = Auto("Škoda")  
  
print(a1.kolesa, a1.znacka) # 4 Tesla  
print(a2.kolesa, a2.znacka) # 4 Škoda
```

Premenná triedy = spoločná pre všetky objekty.

Premenná inštancie = unikátna pre každý objekt.

Typické chyby

Zabudnuté **self** v metódach:

```
class Student:  
    def pozdrav():    # chýba self  
        print("Ahoj")
```

Snažiť sa volať metódu triedy bez vytvorenia objektu:

```
s = Student    # to je odkaz na triedu, nie objekt  
s.pozdrav()   # TypeError
```

Mini cvičenia

1. Vytvor triedu **Pes** s atribútmi **meno** a **vek** a metódou **haf()**, ktorá vypíše „Haf!“.
2. Vytvor triedu **Kniha** s atribútmi **nazov**, **autor**. Pridaj metódu **info()**, ktorá vypíše názov a autora.
3. Rozšír triedu **Student** o atribút **znamky** (zoznam) a metódu **pridaj_znamku()**.

Premenné triedy

Inštančné premenné (recap)

- Definujú sa v metóde `__init__`.
- Každý objekt (inštancia) má **svoje vlastné hodnoty**.

```
class Student:  
    def __init__(self, meno, vek):  
        self.meno = meno      # inštančná premenná  
        self.vek = vek        # inštančná premenná  
  
s1 = Student("Eva", 17)  
s2 = Student("Peter", 18)  
  
print(s1.meno)  # Eva  
print(s2.meno)  # Peter
```

Každý študent má iné meno a vek → tieto hodnoty sú uložené v samotných objektoch.

Premenné triedy

- Definujú sa **priamo v triede**, mimo `__init__`.
- Sú **spoločné pre všetky inštancie** triedy.

```
class Student:  
    skola = "Gymnázium"      # premenná triedy  
  
    def __init__(self, meno):  
        self.meno = meno      # inštančná premenná  
  
s1 = Student("Eva")  
s2 = Student("Peter")  
  
print(s1.skola)  # Gymnázium  
print(s2.skola)  # Gymnázium
```

Oboch študentov spája rovnaká hodnota `skola`.

Rozdiel v zmene premenných

Zmena inštančnej premennej

```
s1.meno = "Nina"  
print(s1.meno)  # Nina  
print(s2.meno)  # Peter (nezmenilo sa)
```

Zmena premennej triedy

```
Student.skola = "Obchodná akadémia"
print(s1.skola) # Obchodná akadémia
print(s2.skola) # Obchodná akadémia
```

Zmena premennej triedy sa prejaví **na všetkých objektoch**.

Príklad použitia premenných tried

Počítadlo inštancií

```
class Student:
    pocet_studentov = 0    # premenná triedy

    def __init__(self, meno):
        self.meno = meno
        Student.pocet_studentov += 1

s1 = Student("Eva")
s2 = Student("Peter")

print(Student.pocet_studentov) # 2
```

Každý nový študent zvýši spoločný čítač.

Typické chyby

Pokus meniť premennú triedy cez inštanciu → vytvorí sa nová inštančná premenná!

```
s1.skola = "SOŠ"    # vytvorí sa nová premenná len pre s1
print(s1.skola)      # SOŠ
print(s2.skola)      # Gymnázium
print(Student.skola) # Gymnázium
```

Správne: meniť priamo cez triedu:

```
Student.skola = "SOŠ"
```

Mini cvičenia

1. Vytvor triedu **Auto** s premennou triedy **kolesa = 4**.
 - Vytvor 2 inštancie a skontroluj, že majú 4 kolesá.
2. Zmeň premennú **kolesa** na **6** cez triedu → ako sa to prejaví na všetkých inštanciach?
3. Skús zmeniť **kolesa** cez jednu inštanciu → čo sa stane?
4. Urob triedu **Pes** s čítačom, ktorý sleduje, koľko psov bolo vytvorených.

Dedičnosť (inheritance)

Čo je dedenie?

- Dedenie = mechanizmus, kde **jedna trieda (potomok)** preberá vlastnosti a metódy inej triedy (rodiča).
- Vďaka tomu nemusíme opakovať kód → **znovupoužiteľnosť** a **prehľadnosť**.

Príklad z reálneho sveta:

- Trieda **Auto** = všeobecná šablóna (značka, rok výroby, počet kolies).
- Trieda **ElektroAuto** = špeciálny typ auta → zdedí všetko z **Auto**, ale pridá atribút **kapacita_baterie**.

Základná syntax

```
class Rodič:  
    def __init__(self, meno):  
        self.meno = meno  
  
    def pozdrav(self):  
        print(f"Ahoj, volám sa {self.meno}")  
  
# Potomok dedí z rodiča  
class Potomok(Rodič):  
    pass  
  
osoba = Potomok("Eva")  
osoba.pozdrav() # Ahoj, volám sa Eva
```

class Potomok(Rodič): znamená, že trieda Potomok dedí všetko z Rodič.

Rozširovanie dedenej triedy

```
class Auto:  
    def __init__(self, znacka):  
        self.znacka = znacka  
  
    def info(self):  
        print(f"Auto značky {self.znacka}")  
  
class ElektroAuto(Auto):  
    def __init__(self, znacka, bateria):  
        super().__init__(znacka) # volá konštruktor rodiča  
        self.bateria = bateria  
  
    def info(self):  
        print(f"Elektroauto {self.znacka} s batériou {self.bateria} kWh")
```

Použitie:

```
a1 = Auto("Škoda")
a2 = ElektroAuto("Tesla", 100)

a1.info()    # Auto značky Škoda
a2.info()    # Elektroauto Tesla s batériou 100 kWh
```

`super()` = zavolá konštruktor alebo metódu rodičovskej triedy.

Viacnásobné dedenie

Python podporuje aj dedenie z viacerých tried:

```
class A:
    def metoda_a(self):
        print("Metóda z A")

class B:
    def metoda_b(self):
        print("Metóda z B")

class C(A, B):
    pass

obj = C()
obj.metoda_a()
obj.metoda_b()
```

Toto sa volá **multiple inheritance**, ale treba ho používať opatrne → môže spôsobiť chaos.

Typické chyby

Zabudnuté `super().__init__()` → nevolá sa rodičovský konštruktor, niektoré atribúty neexistujú.

Rovnaké metódy vo viacerých rodičoch pri multiple inheritance → konflikt.

Pokus volať metódu rodiča cez meno triedy namiesto `super()`.

Mini cvičenia

1. Vytvor triedu `Zviera` s metódou `zvuk()`. Vytvor triedy `Pes` a `Mačka`, ktoré túto metódu prepíšu (`haf`, `mňau`).
2. Vytvor triedu `Osoba` (atribúty: meno, vek). Potom vytvor triedu `Student` (atribút: škola), ktorá dedí z `Osoba`.
3. Rozšír triedu `Auto` o `SportoveAuto`, ktoré má navyše atribút `max_rychlosť`.
4. Skús spraviť triedu `HybridAuto`, ktorá dedí z `Auto` aj `ElektroAuto`.

Sekcia XI: Pokročilé OOP

Pokročilé OOP (polymorfizmus, duck typing)

Čo je polymorfizmus?

Polymorfizmus → rovnaká metóda má rôzne správanie podľa triedy objektu.

Príklad:

```
class Pes:
    def zvuk(self):
        return "Haf!"

class Macka:
    def zvuk(self):
        return "Mňau!"

zvierata = [Pes(), Macka()]

for zviera in zvierata:
    print(zviera.zvuk())
```

Výstup:

```
Haf!
Mňau!
```

Rovnaké volanie `zvuk()` → ale výsledok závisí od objektu.

Prečo je polymorfizmus dôležitý?

- Nemusíme sa starať, aký presne typ objektu používame.
- Program je flexibilnejší – môže pracovať s rôznymi objektmi rovnako.
- Jedna metóda → veľa rôznych správaní.

Duck typing

Python má filozofiu:

„Ak to chodí ako kačka a kváka ako kačka, je to kačka.“

Inými slovami: **nezáleží na tom, z akej triedy objekt je, ale aké má metódy/atribúty.**

Príklad:

```
class Kacica:  
    def zvuk(self):  
        return "Kvak!"  
  
class Robot:  
    def zvuk(self):  
        return "Píííp!"  
  
def urob_zvuk(objekt):  
    print(objekt.zvuk())  
  
urob_zvuk(Kacica()) # Kvak!  
urob_zvuk(Robot()) # Píííp!
```

Nezáleží, že Robot nie je „zviera“ – má metódu `zvuk()`, a to Pythonu stačí.

Prepisovanie metód (method overriding)

Potomok môže **prepísat** metódu rodiča → to je jeden zo spôsobov polymorfizmu.

```
class Zviera:  
    def zvuk(self):  
        return "?"  
  
class Pes(Zviera):  
    def zvuk(self):  
        return "Haf!"  
  
class Macka(Zviera):  
    def zvuk(self):  
        return "Mňau!"
```

Typické chyby

Očakávanie, že polymorfizmus funguje, keď objekty nemajú spoločné metódy.

```
class Pes:  
    def haf(self): return "Haf!"  
  
class Macka:  
    def mnau(self): return "Mňau!"  
  
for zviera in [Pes(), Macka()]:  
    print(zviera.zvuk()) # AttributeError
```

Riešenie: metódy musia mať rovnaký názov (`zvuk`).

Prílišné spoliehanie sa na typy → Python nepotrebuje deklarácie typov pre polymorfizmus.

Mini cvičenia

1. Vytvor triedy `Pes`, `Macka`, `Kohut` – všetky s metódou `zvuk()`. Prejdi cez ne v zozname a zavolaj `zvuk()`.
2. Vytvor funkciu `spusti(objekt)`, ktorá zavolá metódu `start()` na akomkoľvek objekte. Otestuj ju na triedach `Auto` a `Notebook`.
3. Ukáž prepísanie metódy: trieda `Osoba` má metódu `info()`. Trieda `Student` prepíše `info()`, aby vypísala aj školu.

Statické a triedne metódy

Bežné (inštančné) metódy – recap

Doteraz všetky metódy pracovali s konkrétnou inštanciou (objektom). Preto vždy mali parameter `self`.

```
class Student:  
    def __init__(self, meno):  
        self.meno = meno  
  
    def pozdrav(self):  
        print(f"Ahoj, som {self.meno}")
```

Použitie:

```
s = Student("Eva")  
s.pozdrav()
```

Metóda `pozdrav()` potrebuje vedieť, s ktorým študentom pracuje.

Statické metódy (@staticmethod)

- Nepotrebujú `self` ani `cls`.
- Sú to „obyčajné“ funkcie, ktoré **len žijú vo vnútri triedy**, aby dávali logický zmysel.
- Hodí sa pre pomocné výpočty.

```
class Matika:  
    @staticmethod  
    def sucet(a, b):  
        return a + b  
  
    print(Matika.sucet(3, 5)) # 8
```

Statická metóda **nevie o žiadnych atribútoch triedy ani objektu**.

Triedne metódy (@classmethod)

- Nepotrebujú `self`, ale majú parameter `cls`.
- Pracujú s triedou ako celkom (napr. so **spoločnými premennými triedy**).

```
class Student:  
    pocet = 0    # premenná triedy  
  
    def __init__(self, meno):  
        self.meno = meno  
        Student.pocet += 1  
  
    @classmethod  
    def kolko_studentov(cls):  
        return f"Počet študentov: {cls.pocet}"  
  
s1 = Student("Eva")  
s2 = Student("Peter")  
  
print(Student.kolko_studentov()) # Počet študentov: 2
```

Triedna metóda pracuje s triedou ako celkom, nie s konkrétnou inštanciou.

Kedy použiť čo?

- **Inštančná metóda** → potrebuje informácie o konkrétnom objekte.
 - **Triedna metóda** → potrebuje informácie o triede ako celku (napr. štatistiky).
 - **Statická metóda** → pomocná funkcia, ktorá súčasne súvisí s triedou, ale nepotrebuje nič z nej.
-

Typické chyby

Zabudnutá dekorácia `@staticmethod` alebo `@classmethod`.

Pokus použiť `self` v statickej metóde → chyba.

Zámena medzi `class` a `static`:

- `@classmethod` má vždy parameter `cls`.
 - `@staticmethod` nemá žiadny implicitný parameter.
-

Mini cvičenia

1. Vytvor triedu `Konvertor` so statickou metódou `c_to_f(c)` → prevedie stupne °C na °F.
2. Vytvor triedu `Auto` s premennou triedy `vyrobene = 0` a triednou metódou `pocet_aut()`, ktorá vráti počet vytvorených áut.
3. Urob triedu `Kalkulacka` so statickými metódami `sucet(a, b)`, `rozdiel(a, b)`, `nasobenie(a, b)`, `delenie(a, b)`.
4. Vytvor triedu `Student` so zoznamom všetkých mien v premennej triedy a triednou metódou `vsetci()`.

Magické metódy

Čo sú magické metódy?

- Magické (alebo špeciálne, dunder) metódy = metódy v Pythone, ktoré majú tvar `__nazov__`.
- Umožňujú, aby sa triedy správali ako vstavané typy (`int`, `list`, ...).
- Python ich volá automaticky v určitých situáciách → ty definuješ, čo sa má stať.

Príklady, ktoré už poznáš:

- `__init__` → konštruktor (pri vytvorení objektu).
- `__str__` → ako sa objekt zobrazí, keď použijeme `print()`.

Najčastejšie používané magické metódy

`__init__` – inicializácia

```
class Student:  
    def __init__(self, meno, vek):  
        self.meno = meno  
        self.vek = vek
```

`__str__` – „ľudský“ výpis

```
class Student:  
    def __init__(self, meno, vek):  
        self.meno = meno  
        self.vek = vek  
  
    def __str__(self):  
        return f"{self.meno}, {self.vek}"  
  
s = Student("Eva", 17)  
print(s) # Eva, 17
```

Bez `__str__` by Python vypísal niečo ako `<__main__.Student object at 0x7f...>`.

`__repr__` – oficiálny výpis (pre vývojárov)

```
def __repr__(self):  
    return f"Student(meno={self.meno}, vek={self.vek})"
```

`__len__` – dĺžka objektu

```
class Trieda:  
    def __init__(self, studenti):  
        self.studenti = studenti  
  
    def __len__(self):  
        return len(self.studenti)  
  
trieda = Trieda(["Eva", "Peter"])  
print(len(trieda)) # 2
```

__add__ – operátor +

```
class Vektor:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __add__(self, other):  
        return Vektor(self.x + other.x, self.y + other.y)  
  
    def __str__(self):  
        return f"({self.x}, {self.y})"  
  
v1 = Vektor(1, 2)  
v2 = Vektor(3, 4)  
print(v1 + v2) # (4, 6)
```

Ďalšie užitočné magické metódy

- __eq__(self, other) → ==
- __lt__(self, other) → <
- __getitem__(self, index) → správanie ako zoznam
- __call__(self, ...) → správanie ako funkcia
- __iter__(self) → iterovanie cez objekt

Typické chyby

Zabudnuté __str__ → výpis objektu je nečitateľný.

Preťaženie operátorov (__add__, __eq__) bez toho, aby to dávalo logiku.

Myslieť si, že __repr__ a __str__ sú to isté – __repr__ je určený pre vývojárov, __str__ pre používateľov.

Mini cvičenia

1. Vytvor triedu Osoba, ktorá má __str__, aby sa pekne vypísala ("Meno, vek").
2. Vytvor triedu ZoznamMien, ktorá má metódu __len__, aby len(objekt) vrátil počet mien.
3. Urob triedu Vektor s metódami __add__ a __sub__ (sčítanie a odčítanie).
4. Skús __eq__, aby sa dalo porovnať, či majú dvaja študenti rovnaké meno.

Sekcia XII: Výnimky, súborový systém a dekorátory

@property

Problém bez @property

V OOP často chceme kontrolovať, ako sa pracuje s atribútmi.

Napríklad trieda Osoba:

```
class Osoba:  
    def __init__(self, meno, vek):  
        self.meno = meno  
        self.vek = vek
```

Problém: nič nebráni tomu, aby niekto nastavil nezmyselný vek:

```
o = Osoba("Eva", -5) # negatívny vek?
```

Klasické riešenie v iných jazykoch: používať „gettery“ a „setters“.

```
class Osoba:  
    def __init__(self, meno, vek):  
        self._vek = vek  
        self.meno = meno  
  
    def get_vek(self):  
        return self._vek  
  
    def set_vek(self, vek):  
        if vek >= 0:  
            self._vek = vek  
        else:  
            print("Vek nemôže byť záporný!")
```

Použitie:

```
o = Osoba("Eva", 20)  
o.set_vek(-5) # kontrola
```

Problém: v Pythone to pôsobí zbytočne ťažkopádne.

Elegantné riešenie: @property

Python umožňuje použiť dekorátor `@property`, aby sme mohli písat:

- **ako keby sme priamo čítali atribút,**
- ale zároveň za tým beží funkcia (ktorá vie urobiť kontrolu).

```
class Osoba:  
    def __init__(self, meno, vek):  
        self._vek = vek  
        self.meno = meno  
  
    @property  
    def vek(self):  
        return self._vek  
  
    @vek.setter  
    def vek(self, hodnota):  
        if hodnota >= 0:  
            self._vek = hodnota  
        else:  
            print("Vek nemôže byť záporný!")
```

Použitie:

```
o = Osoba("Eva", 20)  
print(o.vek)    # čítanie → volá getter  
  
o.vek = 25      # volá setter  
o.vek = -5      # "Vek nemôže byť záporný!"
```

Takto máme čitateľný kód (`o.vek = 25`) a zároveň bezpečnosť.

Read-only vlastnosti

Ak nenapíšeme `@vek.setter`, atribút je **len na čítanie**.

```
class Kruh:  
    def __init__(self, r):  
        self.r = r  
  
    @property  
    def obvod(self):  
        return 2 * 3.14 * self.r
```

Použitie:

```
k = Kruh(10)
print(k.obvod)    # 62.8
k.obvod = 100    # chyba → obvod je len na čítanie
```

Typické chyby

Zabudnutý dekorátor → metóda sa volá ako funkcia, nie ako atribút.

Premenovanie → ak máš metódu aj atribút s rovnakým menom, môže vzniknúť konflikt.

Nesprávne použitie `@property` na funkcie, ktoré majú vedľajšie efekty (property by mali byť „bezpečné“).

Mini cvičenia

1. Vytvor triedu `Auto` s atribútom `rychlosť`. Zabráň tomu, aby sa dala nastaviť záporná hodnota.
2. Urob triedu `Zamestnanec` s property `plat` → ak nieko zadá menej ako minimálnu mzdu, nastav ho na minimálnu mzdu.
3. Skús triedu `Kruh`, ktorá má property `plocha` (len na čítanie).
4. Vytvor triedu `Student`, ktorá má property `meno` → meno sa pri nastavení vždy automaticky naformátuje s veľkým prvým písmenom.

Dekorátory

Čo je dekorátor?

Dekorátor je **funkcia, ktorá zoberie inú funkciu ako vstup a vráti ju vylepšenú.**

Predstav si to ako keby si mal hamburger:

- Máso = pôvodná funkcia.
- Dekorátor = žemľa, omáčka, syr → pridá niečo navyše, bez zmeny mäsa.

Základný príklad dekorátora

```
- □ ×  
def dekorator(funkcia):  
    def wrapper():  
        print("Pred funkciovou")  
        funkcia()  
        print("Po funkcii")  
    return wrapper  
  
@dekorator  
def ahoj():  
    print("Ahoj svet!")  
  
ahoj()
```

Výstup:

```
- □ ×  
Pred funkciovou  
Ahoj svet!  
Po funkcii
```

@dekorator je len skrátený zápis pre:

```
- □ ×  
ahoj = dekorator(ahoj)
```

Dekorátor s argumentmi

```
def logovanie(funkcia):
    def wrapper(*args, **kwargs):
        print(f"Volám {funkcia.__name__} s argumentmi {args} {kwargs}")
        vysledok = funkcia(*args, **kwargs)
        print(f"Výsledok: {vysledok}")
        return vysledok
    return wrapper

@logovanie
def suiset(a, b):
    return a + b

suiset(3, 5)
```

Výstup:

```
Volám suiset s argumentmi (3, 5) {}
Výsledok: 8
```

Používame `*args` a `**kwargs`, aby dekorátor fungoval s akokoľvek funkciou.

Reálne využitie dekorátorov

- **Logovanie** (zaznamenávanie volaní funkcií).
- **Meranie výkonu** (časovanie funkcie).
- **Autorizácia** (kontrola, či má používateľ právo spustiť funkciu).
- **Cacheovanie** výsledkov (`functools.lru_cache`).
- **Webové frameworky** (Flask, Django – dekorátory určujú, ktorá funkcia je „route“ na stránke).

Príklad: meranie času

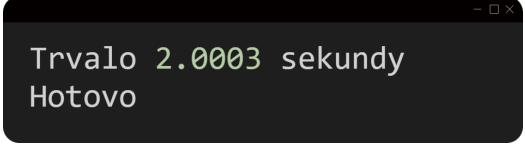
```
import time

def timer(funkcia):
    def wrapper(*args, **kwargs):
        start = time.time()
        vysledok = funkcia(*args, **kwargs)
        end = time.time()
        print(f"Trvalo {end - start:.4f} sekundy")
        return vysledok
    return wrapper

@timer
def pomala_funkcia():
    time.sleep(2)
    return "Hotovo"

print(pomala_funkcia())
```

Výstup:



```
Trvalo 2.0003 sekundy
Hotovo
```

Typické chyby

Zabudnuté `return` vo wrapperi → pôvodná funkcia nič nevracia.

Dekorátor neberie `*args, **kwargs` → funguje len pre funkcie s fixnými parametrami.

Viacnásobné dekorátory v nesprávnom poradí → mení sa logika.

Mini cvičenia

1. Napíš dekorátor `pozdrav_pred`, ktorý vždy pred funkciou vypíše „Vitaj!“.
2. Napíš dekorátor `over_heslo`, ktorý pred spustením funkcie skontroluje, či je heslo správne.
3. Vytvor dekorátor `opakuj(n)`, ktorý spustí funkciu `n`-krát.
4. Skús dekorátor, ktorý premení výsledok funkcie na veľké písmená (`upper()`).

Výnimky (exception handling)

Prečo potrebujeme výnimky?

Predstav si tento kód:

```
a = int(input("Zadaj číslo: "))
b = int(input("Zadaj ďalšie číslo: "))
print(a / b)
```

Čo sa stane, ak:

- používateľ zadá písmeno namiesto čísla? → `ValueError`
- alebo zadá 0 ako druhé číslo? → `ZeroDivisionError`

Program sa **hned' zastaví**.

Ale v reálnych aplikáciach (bankovníctvo, hry, weby) → **musí pokračovať ďalej**.

Základná štruktúra try/except

```
try:
    # kód, ktorý môže vyhodiť chybu
    x = int("abc")
except:
    print("Chyba pri prevode!")
```

Výstup:

```
Chyba pri prevode!
```

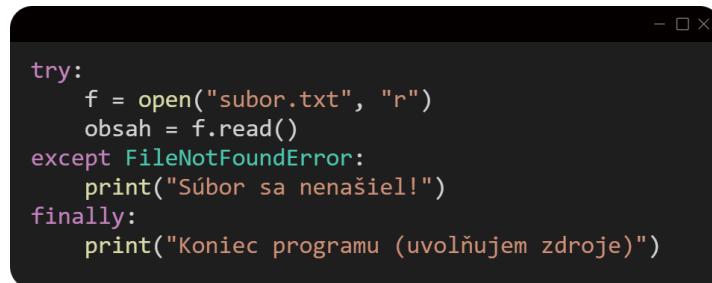
Program nespadol, chyba sa zachytila.

Konkrétne typy chýb

```
try:
    a = int(input("Zadaj číslo: "))
    b = int(input("Zadaj ďalšie číslo: "))
    print(a / b)
except ValueError:
    print("Musíš zadat číslo!")
except ZeroDivisionError:
    print("Nesmieš deliť nulou!")
```

Môžeme zachytiť rôzne typy výnimiek osobitne.

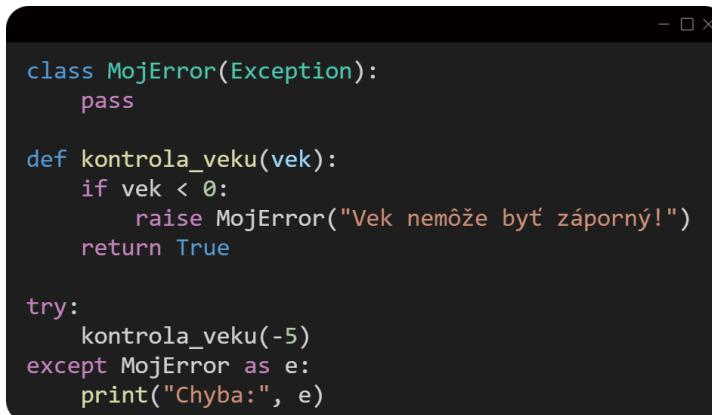
Blok finally



```
try:
    f = open("subor.txt", "r")
    obsah = f.read()
except FileNotFoundError:
    print("Súbor sa nenašiel!")
finally:
    print("Koniec programu (uvolňujem zdroje)")
```

- `finally` sa vykoná vždy → bez ohľadu na to, či nastala chyba.
-

Vlastné výnimky



```
class MojError(Exception):
    pass

def kontrola_veku(vek):
    if vek < 0:
        raise MojError("Vek nemôže byť záporný!")
    return True

try:
    kontrola_veku(-5)
except MojError as e:
    print("Chyba:", e)
```

Môžeme si vytvoriť aj **vlastnú výnimku**.

Typické chyby

Používanie prázdnego `except:` bez špecifikácie → schová všetky chyby, tāžko sa debuguje.

Zabudnuté `finally` pri práci so súbormi → súbor zostane otvorený.

Prehnané používanie výnimiek namiesto jednoduchých podmienok (`if`).

Mini cvičenia

1. Napíš program, ktorý sa spýta na dve čísla a vydelí ich → ošetri `ValueError` a `ZeroDivisionError`.
2. Vytvor funkciu, ktorá sa spýta na vek → ak nie je celé číslo alebo je menšie než 0, vypíš chybu.
3. Skús si vytvoriť vlastnú výnimku `NegativeNumberError` a použi ju vo funkciu, ktorá prijíma iba kladné čísla.
4. Urob program, ktorý otvorí súbor a vypíše obsah → ošetri `FileNotFoundException`.

Práca so súbormi

Otváranie súboru na zápis

Funkcia `open(cesta, mód)` → otvorí súbor.

Najčastejšie módy pre zápis:

- `"w"` → write → prepíše súbor (ak neexistuje, vytvorí).
- `"a"` → append → pridáva na koniec (ak neexistuje, vytvorí).

```
# mód w - prepíše celý súbor
with open("data.txt", "w") as f:
    f.write("Prvý riadok\n")

# mód a - pridáva na koniec
with open("data.txt", "a") as f:
    f.write("Druhý riadok\n")
```

Vďaka `with` sa súbor automaticky zatvorí (`close()`).

Čítanie + zapisovanie (kombinované módy)

- `"r+"` → čítanie aj zapisovanie (súbor musí existovať).
- `"w+"` → zapisovanie + čítanie (súbor sa vytvorí, starý sa zmaže).
- `"a+"` → pridávanie + čítanie (záznamy sa pridávajú na koniec).

Typické chyby

Použitie `"w"` → nevedomky prepíše existujúci súbor.

Zabudnuté zatvoriť súbor → vďaka `with open(...)` sa tomu vyhneme.

Pokus zapisovať do súboru, na ktorý nemáme práva → `PermissionError`.

Práca so súbormi bez spracovania výnimiek → program padne.

Spracovanie chýb

```
try:
    with open("data.txt", "w") as f:
        f.write("Nový obsah")
except FileNotFoundError:
    print("Súbor sa nenašiel!")
except PermissionError:
    print("Nemáš právo zapisovať do tohto súboru.")
except Exception as e:
    print("Neočakávaná chyba:", e)
```

Mini cvičenia

1. Vytvor program, ktorý zapíše do súboru `znamky.txt` päť známok.
2. Rozšír program tak, aby každé spustenie pridalo známku na koniec (mód `"a"`).
3. Urob program, ktorý skontroluje, či súbor existuje. Ak áno → pridaj nový záznam, ak nie → vytvor súbor s prvým záznamom.
4. Skús otvoriť súbor bez povolenia (napr. systémový súbor) a ošetri chybu `PermissionError`.

Detekcia súborov

Prečo kontrolovať súbor?

Ak sa pokúsiš otvoriť súbor, ktorý neexistuje:

```
f = open("data.txt", "r") # chyba, ak súbor neexistuje
```

Python vyhodí chybu:

```
FileNotFoundException
```

Preto je dobré najskôr skontrolovať, či súbor **existuje**.

Použitie `os.path.exists()`

```
import os

cesta = "data.txt"

if os.path.exists(cesta):
    print("Súbor existuje.")
else:
    print("Súbor neexistuje.")
```

Kontrola, či je to súbor alebo priečinok

```
import os

cesta = "moj_priečinok"

if os.path.isfile(cesta):
    print("Je to súbor")
elif os.path.isdir(cesta):
    print("Je to priečinok")
else:
    print("Neexistuje")
```

Použitie s výnimkami

Najlepšia prax je kombinovať **kontrolu existencie** so **spracovaním výnimky**:

```
import os

cesta = "data.txt"

if os.path.exists(cesta):
    try:
        with open(cesta, "r") as f:
            obsah = f.read()
            print("Obsah súboru:", obsah)
    except Exception as e:
        print("Chyba pri čítaní:", e)
else:
    print("Súbor neexistuje.")
```

Typické chyby

Zabudnutý import: `import os`.

Používanie `os.path.exists()` bez uistenia, či je to súbor alebo priečinok.

Mysliť si, že `exists()` = „súbor je čitateľný“ → nie, súbor môže existovať, ale nemusíme mať práva na čítanie.

Mini cvičenia

1. Napíš program, ktorý sa spýta na cestu k súboru a vypíše, či existuje.
2. Skontroluj, či existuje súbor `data.txt`, a ak áno, vypíš jeho obsah.
3. Vytvor funkciu `bezpecne_otvor(cesta)`, ktorá:
 - najprv skontroluje existenciu,
 - ak súbor existuje → otvorí ho,
 - inak vypíše chybové hlásenie.
4. Skús skontrolovať, či súbor `subor.txt` je súbor alebo priečinok.

EXKLUZÍVNY KUPÓN: SURVIVE2025

Tento kód je iba pre prvých 200, čo dočítali Python Survival Guide až do konca.

Znamená to, že to s Pythonom myslíš vážne, a prto ti dávame odmenu:

- 9€ zľava na kurz **Python 1.0 - Základy**
- prístup k projektom, bonusom a mentoringu, ktoré v PDF nenájdeš
- ponuka platí len pre študentov, ktorí chcú urobiť svoj **prvý veľký krok do IT**

Uplatni kód **SURVIVE2025** pri registrácii a začni kurz za polovičnú cenu ešte dnes.

