

OMM

DATOVÉ TYPY V PYTHONU

Andrej Pčelovodov

OMM

DATOVÉ TYPY V PYTHONU

bool • int • float • string

Andrej Pčelovodov

OMM

DATOVÉ TYPY V PYTHONU

bool • int • float • string

list • tuple • dict • set

CONDITIONALS & ITERATION

ŘÍZENÍ TOKU PROGRAMU V PYTHONU

Andrej Pčelovodov

CONDITIONALS & ITERATION

ŘÍZENÍ TOKU PROGRAMU V PYTHONU

if • match • for • while • itertools

Andrej Pčelovodov

PROČ JE TO DŮLEŽITÉ?

Andrej Pčelovodov

PROČ JE TO DŮLEŽITÉ?

- Rozhodování (podmínky)

PROČ JE TO DŮLEŽITÉ?

- Rozhodování (podmínky)
- Opakování práce (cykly)

PROČ JE TO DŮLEŽITÉ?

- Rozhodování (podmínky)
- Opakování práce (cykly)
- Čistý a čitelný kód

IF / ELIF / ELSE

```
1 age = 18
2
3 if age >= 18:
4     print("Dospělý")
5 elif age >= 15:
6     print("Téměř")
7 else:
8     print("Dítě")
```

VNOŘENÉ PODMÍNKY (NESTED IF)



```
1 if user_logged:  
2     if is_admin:  
3         print("Admin panel")
```

VNOŘENÉ PODMÍNKY (NESTED IF)



```
1 if user_logged:  
2     if is_admin:  
3         print("Admin panel")
```

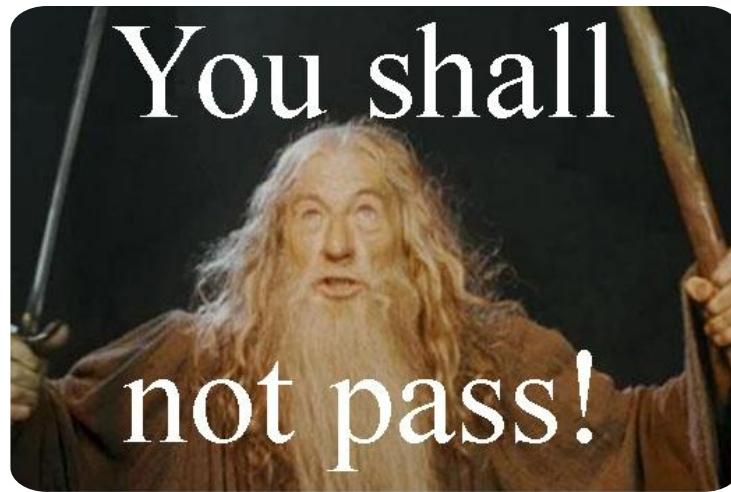
👉 raději zjednodušovat a používat **guard clauses!!!**

VNOŘENÉ PODMÍNKY (NESTED IF)



```
1 if user_logged:  
2     if is_admin:  
3         print("Admin panel")
```

👉 raději zjednodušovat a používat **guard clauses!!!**



Andrej Pčelovodov

VNOŘENÉ PODMÍNKY (NESTED IF)



```
1 if user_logged:  
2     if is_admin:  
3         print("Admin panel")
```

👉 raději zjednodušovat a používat **guard clauses!!!**

VNOŘENÉ PODMÍNKY (NESTED IF)



```
1 if user_logged:  
2     if is_admin:  
3         print("Admin panel")
```

👉 raději zjednodušovat a používat **guard clauses!!!**

```
1 if not user_logged and not is_admin:  
2     return  
3  
4 print("Admin panel")
```

TERNÁRNÍ OPERÁTOR

```
1 status = "OK" if value > 0 else "FAIL"
```

TERNÁRNÍ OPERÁTOR

```
1 status = "OK" if value > 0 else "FAIL"
```

```
1 n = -5
2
3 res = "Positive" if n > 0 else "Negative" if n < 0 else "Zero"
```

TERNÁRNÍ OPERÁTOR

```
1 status = "OK" if value > 0 else "FAIL"
```

```
1 n = -5
2
3 res = "Positive" if n > 0 else "Negative" if n < 0 else "Zero"
```

Používat jen pro jednoduché výrazy

PATTERN MATCHING (MATCH)

```
1 cmd = "start"
2
3 match cmd:
4     case "start":
5         run()
6     case "stop":
7         stop()
8     case _:
9         print("Unknown")
```

PATTERN MATCHING (MATCH)

```
1 cmd = "start"
2
3 match cmd:
4     case "start":
5         run()
6     case "stop":
7         stop()
8     case _:
9         print("Unknown")
```

```
1 x = 20
2
3 match x:
4     case 10 | 20 | 30: # Matches 10, 20, or 30
5         print(f"Matched: {x}")
6     case _:
7         print("No match found")
```

FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list
tuple
set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```

FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list
tuple
set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```

FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list
tuple
set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```

FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list
tuple
set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```

FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list

tuple

set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```



FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list
tuple
set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```



```
1 for name in ["Alice", "Bob", "Eve"]:  
2     print(name)
```

FOR CYKLUS (LIST)

Pomocí smyčky for můžeme provést sadu příkazů, jednou pro každou položku v:

list
tuple
set atd.

```
1 fruits = ["apple", "banana", "cherry"]  
2 for x in fruits:  
3     print(x)
```



```
1 for name in ["Alice", "Bob", "Eve"]:  
2     print(name)
```



Andrej Pčelovodov

FOR CYKLUS (STRING)

I řetězce jsou iterovatelné objekty (obsahují posloupnost znaků)

```
1 fruit = "banana"  
2  
3 for x in fruit:  
4     print(x)
```

FOR CYKLUS (STRING)

I řetězce jsou iterovatelné objekty (obsahují posloupnost znaků)

```
1 fruit = "banana"  
2  
3 for x in fruit:  
4     print(x)
```

FOR CYKLUS (STRING)

I řetězce jsou iterovatelné objekty (obsahují posloupnost znaků)

```
1 fruit = "banana"  
2  
3 for x in fruit:  
4     print(x)
```

FOR CYKLUS (STRING)

I řetězce jsou iterovatelné objekty (obsahují posloupnost znaků)

```
1 fruit = "banana"  
2  
3 for x in fruit:  
4     print(x)
```

FOR CYKLUS (BREAK)

Pomocí příkazu break můžeme smyčku zastavit dříve, než projde všemi položkami

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     print(x)
4     if x == "banana":
5         break
```

FOR CYKLUS (BREAK)

Pomocí příkazu break můžeme smyčku zastavit dříve, než projde všemi položkami

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     print(x)
4     if x == "banana":
5         break
```

FOR CYKLUS (BREAK)

Pomocí příkazu break můžeme smyčku zastavit dříve, než projde všemi položkami

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     print(x)
4     if x == "banana":
5         break
```

FOR CYKLUS (BREAK)

Pomocí příkazu break můžeme smyčku zastavit dříve, než projde všemi položkami

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     print(x)
4     if x == "banana":
5         break
```

FOR CYKLUS (BREAK)

Pomocí příkazu break můžeme smyčku zastavit dříve, než projde všemi položkami

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     print(x)
4     if x == "banana":
5         break
```

FOR CYKLUS (BREAK)

Pomocí příkazu break můžeme smyčku zastavit dříve, než projde všemi položkami

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     print(x)
4     if x == "banana":
5         break
```

FOR CYKLUS (CONTINUE)

Pomocí příkazu continue můžeme zastavit aktuální iteraci smyčky a pokračovat další.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     if x == "banana":
4         continue
5     print(x)
```

FOR CYKLUS (CONTINUE)

Pomocí příkazu continue můžeme zastavit aktuální iteraci smyčky a pokračovat další.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     if x == "banana":
4         continue
5     print(x)
```

FOR CYKLUS (CONTINUE)

Pomocí příkazu continue můžeme zastavit aktuální iteraci smyčky a pokračovat další.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     if x == "banana":
4         continue
5     print(x)
```

FOR CYKLUS (CONTINUE)

Pomocí příkazu continue můžeme zastavit aktuální iteraci smyčky a pokračovat další.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     if x == "banana":
4         continue
5     print(x)
```

FOR CYKLUS (CONTINUE)

Pomocí příkazu continue můžeme zastavit aktuální iteraci smyčky a pokračovat další.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     if x == "banana":
4         continue
5     print(x)
```

FOR CYKLUS (CONTINUE)

Pomocí příkazu continue můžeme zastavit aktuální iteraci smyčky a pokračovat další.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3     if x == "banana":
4         continue
5     print(x)
```

FOR CYKLUS (RANGE)

Chcete-li provést smyčku přes sadu kódu určitý početkrát, můžete použít funkci range(). Funkce range() vrací posloupnost čísel, která začíná standardně od 0, zvyšuje se o 1 (standardně) a končí u zadaného čísla.

```
1 for x in range(6):  
2     print(x)
```

FOR CYKLUS (RANGE)

Chcete-li provést smyčku přes sadu kódu určitý početkrát, můžete použít funkci range(). Funkce range() vrací posloupnost čísel, která začíná standardně od 0, zvyšuje se o 1 (standardně) a končí u zadaného čísla.

```
1 for x in range(6):
2     print(x)
```

FOR CYKLUS (RANGE)

Chcete-li provést smyčku přes sadu kódu určitý početkrát, můžete použít funkci range(). Funkce range() vrací posloupnost čísel, která začíná standardně od 0, zvyšuje se o 1 (standardně) a končí u zadaného čísla.

```
1 for x in range(6):  
2     print(x)
```

RANGE()

Funkce range() má výchozí počáteční hodnotu 0, ale je možné zadat počáteční hodnotu přidáním parametru: range(1, 5), což znamená hodnoty od 1 do 5 (ale bez 5).

Funkce range() ve výchozím nastavení zvyšuje posloupnost o 1, je však možné zadat hodnotu přírůstku přidáním třetího parametru: range(0, 10, 2):

```
1 range(5)      # 0..4
2 range(1, 5)    # 1..4
3 range(0, 10, 2) # krok 2
```

RANGE()

Funkce range() má výchozí počáteční hodnotu 0, ale je možné zadat počáteční hodnotu přidáním parametru: range(1, 5), což znamená hodnoty od 1 do 5 (ale bez 5).

Funkce range() ve výchozím nastavení zvyšuje posloupnost o 1, je však možné zadat hodnotu přírůstku přidáním třetího parametru: range(0, 10, 2):

```
1 range(5)      # 0..4
2 range(1, 5)    # 1..4
3 range(0, 10, 2) # krok 2
```

RANGE()

Funkce range() má výchozí počáteční hodnotu 0, ale je možné zadat počáteční hodnotu přidáním parametru: range(1, 5), což znamená hodnoty od 1 do 5 (ale bez 5).

Funkce range() ve výchozím nastavení zvyšuje posloupnost o 1, je však možné zadat hodnotu přírůstku přidáním třetího parametru: range(0, 10, 2):

```
1 range(5)      # 0..4
2 range(1, 5)    # 1..4
3 range(0, 10, 2) # krok 2
```

RANGE()

Funkce range() má výchozí počáteční hodnotu 0, ale je možné zadat počáteční hodnotu přidáním parametru: range(1, 5), což znamená hodnoty od 1 do 5 (ale bez 5).

Funkce range() ve výchozím nastavení zvyšuje posloupnost o 1, je však možné zadat hodnotu přírůstku přidáním třetího parametru: range(0, 10, 2):

```
1 range(5)      # 0..4
2 range(1, 5)    # 1..4
3 range(0, 10, 2) # krok 2
```

FOR CYKLUS (ELSE)

Klíčové slovo else ve smyčce for určuje blok kódu, který se má provést po dokončení smyčky.

Blok else NEBUDE proveden, pokud je smyčka zastavena příkazem break.

```
1 for x in range(6):
2     if x == 3: break
3     print(x)
4 else:
5     print("Finally finished!")
```

FOR CYKLUS (ELSE)

Klíčové slovo else ve smyčce for určuje blok kódu, který se má provést po dokončení smyčky.

Blok else NEBUDE proveden, pokud je smyčka zastavena příkazem break.

```
1 for x in range(6):
2     if x == 3: break
3     print(x)
4 else:
5     print("Finally finished!")
```

FOR CYKLUS (ELSE)

Klíčové slovo else ve smyčce for určuje blok kódu, který se má provést po dokončení smyčky.

Blok else NEBUDE proveden, pokud je smyčka zastavena příkazem break.

```
1 for x in range(6):
2     if x == 3: break
3     print(x)
4 else:
5     print("Finally finished!")
```

FOR CYKLUS (ELSE)

Klíčové slovo else ve smyčce for určuje blok kódu, který se má provést po dokončení smyčky.

Blok else NEBUDE proveden, pokud je smyčka zastavena příkazem break.

```
1 for x in range(6):
2     if x == 3: break
3     print(x)
4 else:
5     print("Finally finished!")
```

FOR CYKLUS (ELSE)

Klíčové slovo else ve smyčce for určuje blok kódu, který se má provést po dokončení smyčky.

Blok else NEBUDE proveden, pokud je smyčka zastavena příkazem break.

```
1 for x in range(6):
2     if x == 3: break
3     print(x)
4 else:
5     print("Finally finished!")
```

FOR CYKLUS (ELSE)

Klíčové slovo else ve smyčce for určuje blok kódu, který se má provést po dokončení smyčky.

Blok else NEBUDE proveden, pokud je smyčka zastavena příkazem break.

```
1 for x in range(6):
2     if x == 3: break
3     print(x)
4 else:
5     print("Finally finished!")
```

VNOŘENÝ FOR CYKLUS

Vnořená smyčka je smyčka uvnitř smyčky.
„Vnitřní smyčka“ se provede jednou za každou iteraci „vnější smyčky“:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5     for y in fruits:
6         print(x, y)
```

VNOŘENÝ FOR CYKLUS

Vnořená smyčka je smyčka uvnitř smyčky.
„Vnitřní smyčka“ se provede jednou za každou iteraci „vnější smyčky“:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5     for y in fruits:
6         print(x, y)
```

VNOŘENÝ FOR CYKLUS

Vnořená smyčka je smyčka uvnitř smyčky.
„Vnitřní smyčka“ se provede jednou za každou iteraci „vnější smyčky“:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5     for y in fruits:
6         print(x, y)
```

VNOŘENÝ FOR CYKLUS

Vnořená smyčka je smyčka uvnitř smyčky.
„Vnitřní smyčka“ se provede jednou za každou iteraci „vnější smyčky“:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5     for y in fruits:
6         print(x, y)
```

VNOŘENÝ FOR CYKLUS

Vnořená smyčka je smyčka uvnitř smyčky.
„Vnitřní smyčka“ se provede jednou za každou iteraci „vnější smyčky“:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5     for y in fruits:
6         print(x, y)
```

VNOŘENÝ FOR CYKLUS

Vnořená smyčka je smyčka uvnitř smyčky.
„Vnitřní smyčka“ se provede jednou za každou iteraci „vnější smyčky“:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
3
4 for x in adj:
5     for y in fruits:
6         print(x, y)
```

PASS

```
1 for x in range(100):  
2     pass
```

Příkaz pass se používá jako zástupný symbol pro budoucí kód.

Při provedení příkazu pass se nic nestane,
ale zabráníte tak chybě, když není povolen prázdný kód.

Prázdný kód není povolen ve:

smyčkách

definících funkcí

definících tříd

v příkazech if

PASS

```
1 for x in range(100):  
2     pass
```

Příkaz pass se používá jako zástupný symbol pro budoucí kód.

Při provedení příkazu pass se nic nestane,
ale zabráníte tak chybě, když není povolen prázdný kód.

Prázdný kód není povolen ve:

smyčkách

definících funkcí

definících tříd

v příkazech if

PASS

```
1 for x in range(100):  
2     pass
```

Příkaz pass se používá jako zástupný symbol pro budoucí kód.

Při provedení příkazu pass se nic nestane,
ale zabráníte tak chybě, když není povolen prázdný kód.

Prázdný kód není povolen ve:

smyčkách

definících funkcí

definících tříd

v příkazech if

PASS

```
1 def myfunction():
2     pass
```

```
1 class Person:
2     pass
```

```
1 a = 33
2 b = 200
3
4 if b > a:
5     pass
```

SMYČKA WHILE

Pomocí smyčky while můžeme provádět sadu příkazů,
dokud je podmínka pravdivá.

```
1 i = 1
2 while i < 6:
3     print(i)
4     i += 1
```

Smyčka while vyžaduje, aby byly připraveny příslušné proměnné.

V tomto příkladu musíme definovat indexovací proměnnou i, kterou nastavíme na hodnotu 1.

SMYČKA WHILE

Pomocí smyčky while můžeme provádět sadu příkazů,
dokud je podmínka pravdivá.

```
1 i = 1
2 while i < 6:
3     print(i)
4     i += 1
```

Smyčka while vyžaduje, aby byly připraveny příslušné proměnné.

V tomto příkladu musíme definovat indexovací proměnnou i, kterou nastavíme na hodnotu 1.

Poznámka: nezapomeňte zvyšovat hodnotu i,
jinak bude smyčka pokračovat donekonečna.

Andrej Pčelovodov

SMYČKA WHILE (BREAK)

Pomocí příkazu break můžeme smyčku zastavit,
i když je podmínka while splněna.

```
1 i = 1
2 while i < 6:
3     print(i)
4     if i == 3:
5         break
6     i += 1
```

SMYČKA WHILE (CONTINUE)

Pomocí příkazu continue
můžeme zastavit aktuální iteraci a pokračovat další.

```
1 i = 0
2 while i < 6:
3     i += 1
4     if i == 3:
5         continue
6     print(i)
```

SMYČKA WHILE (ELSE)

Pomocí příkazu else můžeme spustit blok kódu jednou, když podmínka již není splněna.

```
1 i = 1
2 while i < 6:
3     print(i)
4     i += 1
5 else:
6     print("i is no longer less than 6")
```



FOR VS WHILE (ŽIVOTNÍ REALITA)

Rozdíl mezi cykly si ukážeme na jednoduchém příkladu:



FOR VS WHILE (ŽIVOTNÍ REALITA)

Rozdíl mezi cykly si ukážeme na jednoduchém příkladu:

Chození do práce

Andrej Pčelovodov



FOR – PRACUJI DO 65 LET

Andrej Pčelovodov



FOR – PRACUJI DO 65 LET

```
1 beginning_age = 20
2 retirement_age = 65
3
4 for age in range(beginning_age, retirement_age):
5     print(f"Je mi {age} let a jdu do práce.")
6
7 print("Odchod do důchodu 🎉")
```



FOR – PRACUJI DO 65 LET

```
1 beginning_age = 20
2 retirement_age = 65
3
4 for age in range(beginning_age, retirement_age):
5     print(f"Je mi {age} let a jdu do práce.")
6
7 print("Odchod do důchodu 🎉")
```



FOR – PRACUJI DO 65 LET

```
1 beginning_age = 20
2 retirement_age = 65
3
4 for age in range(beginning_age, retirement_age):
5     print(f"Je mi {age} let a jdu do práce.")
6
7 print("Odchod do důchodu 🎉")
```



FOR – PRACUJI DO 65 LET

```
1 beginning_age = 20
2 retirement_age = 65
3
4 for age in range(beginning_age, retirement_age):
5     print(f"Je mi {age} let a jdu do práce.")
6
7 print("Odchod do důchodu 🎉")
```



FOR – PRACUJI DO 65 LET

```
1 beginning_age = 20
2 retirement_age = 65
3
4 for age in range(beginning_age, retirement_age):
5     print(f"Je mi {age} let a jdu do práce.")
6
7 print("Odchod do důchodu 🎉")
```



FOR – PRACUJI DO 65 LET

```
1 beginning_age = 20
2 retirement_age = 65
3
4 for age in range(beginning_age, retirement_age):
5     print(f"Je mi {age} let a jdu do práce.")
6
7 print("Odchod do důchodu 🎉")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```



WHILE – PRACUJI DOKUD ŽIJU

```
1 age = 20
2 alive = True
3
4 while alive:
5     print(f"Je mi {age} let a jdu do práce.")
6     age += 1
7
8     if age == 100:
9         alive = False
10
11 print("Konec života.")
```

PŘÍKLAD Z PRAXE

The source code of Windows' troubleshooting program has leaked

```
1 #include <windows.h>
2 #include <stdio.h>
3
4 int main() {
5     printf("Searching for problems...\n");
6     Sleep(60000);
7     printf("We didn't find any problems\n");
8 }
9
```

PŘÍKLAD Z PRAXE

Andrej Pčelovodov

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

PŘÍKLAD Z PRAXE

```
1 import time
2
3 print("Starting Windows Troubleshooter...")
4
5 start_time = time.time()
6 duration = 60 # jak dlouho budeme 'hledat problém' (v sekundách)
7
8 while time.time() - start_time < duration:
9     print("Searching for problems...")
10    time.sleep(2)
11
12 print("We didn't find any problems.")
```

WALRUS OPERÁTOR :=



Andrej Pčelovodov

WALRUS OPERÁTOR :=

Andrej Pčelovodov

WALRUS OPERÁTOR :=

```
1 while (line := file.readline()):  
2     print(line)
```

WALRUS OPERÁTOR :=

```
1 d = [
2     {"userId": 1, "name": "rahul", "completed": False},
3     {"userId": 1, "name": "rohit", "completed": False},
4     {"userId": 1, "name": "ram", "completed": False},
5     {"userId": 1, "name": "ravan", "completed": True}
6 ]
7
8 print("With Python 3.8 Walrus Operator:")
9 for entry in d:
10     if name := entry.get("name"):
11         print(name)
12
13 print("Without Walrus operator:")
14 for entry in d:
15     name = entry.get("name")
```

Andrej Pčelovodov

ITERABLE VS ITERATOR

Seznamy (list), tuple, slovníky (dict) a množiny (set) jsou všechny iterovatelné objekty.

Jedná se o **iterovatelné kontejnery**, ze kterých můžete získat iterátor.

Všechny tyto objekty mají metodu `__iter__()`, která se používá k získání iterátoru.

Iterátor je definován jako typ objektu, který obsahuje hodnoty, ke kterým lze přistupovat nebo je iterovat pomocí smyčky.

ITERABLE VS ITERATOR

Seznamy (list), tuple, slovníky (dict) a množiny (set) jsou všechny iterovatelné objekty.

Jedná se o **iterovatelné kontejnery**, ze kterých můžete získat iterátor.

Všechny tyto objekty mají metodu `__iter__()`, která se používá k získání iterátoru.

Iterátor je definován jako typ objektu, který obsahuje hodnoty, ke kterým lze přistupovat nebo je iterovat pomocí smyčky.

- Iterable → lze projít (list, str, dict)

ITERABLE VS ITERATOR

Seznamy (list), tuple, slovníky (dict) a množiny (set) jsou všechny iterovatelné objekty.

Jedná se o **iterovatelné kontejnery**, ze kterých můžete získat iterátor.

Všechny tyto objekty mají metodu `__iter__()`, která se používá k získání iterátoru.

Iterátor je definován jako typ objektu, který obsahuje hodnoty, ke kterým lze přistupovat nebo je iterovat pomocí smyčky.

- Iterable → lze projít (list, str, dict)
- Iterator → má `__iter__()` a `__next__()` metody

ITERABLE VS ITERATOR

```
1 mytuple = ("apple", "banana", "cherry")
2 myit = iter(mytuple)
3
4 print(next(myit))
5 print(next(myit))
6 print(next(myit))
```

ITERABLE VS ITERATOR

I řetězce jsou iterovatelné objekty a mohou vracet iterátor

```
1 mystr = "banana"
2 myit = iter(mystr)
3
4 print(next(myit))
5 print(next(myit))
6 print(next(myit))
7 print(next(myit))
8 print(next(myit))
9 print(next(myit))
```

PROCHÁZENÍ ITERÁTOREM

Můžeme také použít smyčku for k iterování přes iterovatelný objekt

```
1 mytuple = ("apple", "banana", "cherry")
2
3 for x in mytuple:
4     print(x)
```

```
1 mystr = "banana"
2
3 for x in mystr:
4     print(x)
```

VÍCE SEKVENCÍ NAJEDNOU

```
1 names = ["A", "B"]
2 scores = [10, 20]
3
4 for n, s in zip(names, scores):
5     print(n, s)
```

ITERTOOLS

Andrej Pčelovodov

ITERTOOLS

- chain

ITERTOOLS

- chain
- count

ITERTOOLS

- chain
- count
- cycle

ITERTOOLS

- chain
- count
- cycle
- islice

ITERTOOLS

- chain
- count
- cycle
- islice

Python Itertools jsou skvělým způsobem, jak vytvářet komplexní iterátory, které pomáhají zrychlit execution time  a psát efektivní kód z hlediska paměti .

ITERTOOLS.CHAIN

Jedná se o funkci, která přijímá řadu iterovatelných objektů a vrací jeden iterovatelný objekt.

Seskupuje všechny iterovatelné objekty dohromady a jako výstup vytvoří jeden iterovatelný objekt.

Jeho výstup nelze použít přímo, a proto je nutné jej explicitně převést na iterovatelné objekty.

```
1 from itertools import chain
2
3 # a list of odd numbers
4 odd = [1, 3, 5, 7, 9]
5
6 # a list of even numbers
7 even = [2, 4, 6, 8, 10]
8
9 # chaining odd and even numbers
10 numbers = list(chain(odd, even))
11
12 print(numbers) # [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]
```

ITERTOOLS.COUNT

itertools.count() se obecně používá s map() k generování po sobě jdoucích data pointů.

Lze jej také použít se zip k přidání sekvencí předáním count jako parametru.

```
1 from itertools import count
2
3 # creates a count iterator object
4 iterator = (count(start = 0, step = 2))
5
6 # prints a odd list of integers
7 print("Even list:",
8      list(next(iterator) for _ in range(5)))
9
10 # creates a count iterator object
11 iterator = (count(start = 1, step = 2))
12
13 # prints a odd list of integers
14 print("Odd list:",
15      list(next(iterator) for _ in range(5)))
```

Andrej Přelovodov list(next(iterator) for _ in range(5))

ITERTOOLS.CYCLE

- Funkce přijímá pouze jeden argument jako vstup, který může být například list, string, tuple atd.
- Funkce vrací objekt typu iterátor.
- V implementaci funkce je návratovým typem yield, který pozastaví provádění funkce bez zničení lokálních proměnných. Používá se generátory, které produkují mezivýsledky.
- Prochází každý prvek ve vstupním argumentu, vrací jej, opakuje cyklus a produkuje nekonečnou posloupnost argumentu.

```
1 from itertools import cycle
2
3 x = cycle([1,2,3])
4 for i in x:
5     print(i)
```

ITERTOOLS.ISLICE

Tento iterátor selektivně vrací hodnoty uvedené v jeho iterovatelném kontejneru předaném jako argument.

```
1 islice(iterable, start, stop, step)
```

```
1 from itertools import islice
2
3 for i in islice(range(20), 1, 5):
4     print(i) # 1\n2\n3\n4
```

KOMBINATORICKÉ GENERÁTORY

```
1 from itertools import permutations, combinations  
2  
3 permutations([1,2,3])  
4 combinations([1,2,3], 2)
```

PERMUTATIONS

Generuje všechny možné uspořádané kombinace dané iterovatelné hodnoty
(například list, string nebo tuple).

Na rozdíl od combinations, kde na pořadí nezáleží, permutace zohledňují pořadí prvků.

Vrací iterátor, který vytváří tuple,
z nichž každý představuje jedinečnou permutaci vstupních prvků.

```
1 from itertools import permutations
2 companyName = "RITE"
3
4 for j in permutations(companyName, 2):
5     print(j)
```

PERMUTATIONS

```
1 # ('R', 'I')
2 # ('R', 'T')
3 # ('R', 'E')
4 # ('I', 'R')
5 # ('I', 'T')
6 # ('I', 'E')
7 # ('T', 'R')
8 # ('T', 'I')
9 # ('T', 'E')
10 # ('E', 'R')
11 # ('E', 'I')
12 # ('E', 'T')
```

COMBINATIONS

Se používá k generování všech možných kombinací zadané délky z daného iterovatelného objektu (například list, string nebo tuple).

Na rozdíl od permutací, kde na pořadí záleží, se combinations zaměřuje pouze na výběr prvků, což znamená, že pořadí není důležité.

Vrací iterátor produkující tuple, z nichž každý představuje jedinečnou kombinaci vstupních prvků.

```
1 from itertools import combinations  
2 companyName = "RITE"  
3  
4 for j in combinations(companyName, 2):  
5     print(j)
```

Andrej Pčelovodov

COMBINATIONS

```
1 # ('R', 'I')
2 # ('R', 'T')
3 # ('R', 'E')
4 # ('I', 'T')
5 # ('I', 'E')
6 # ('T', 'E')
```

COMBINATIONS

```
1 # ('R', 'I')
2 # ('R', 'T')
3 # ('R', 'E')
4 # ('I', 'T')
5 # ('I', 'E')
6 # ('T', 'E')
```



Protože💡 ('R', 'I') == ('I', 'R')

SHRNUTÍ

Andrej Pčelovodov

SHRNUTÍ

- if / match → rozhodování

SHRNUTÍ

- if / match → rozhodování
- for / while → opakování

SHRNUTÍ

- if / match → rozhodování
- for / while → opakování
- itertools → efektivita, výkon a elegance

DOMÁCÍ ÚKOL

Andrej Pčelovodov

DOMÁCÍ ÚKOL

Nainstalovat:

Andrej Pčelovodov

DOMÁCÍ ÚKOL

Nainstalovat:

1. [Eclipse Adoptium JDK 25](#)

DOMÁCÍ ÚKOL

Nainstalovat:

1. [Eclipse Adoptium JDK 25](#)
2. [Notepad++](#)

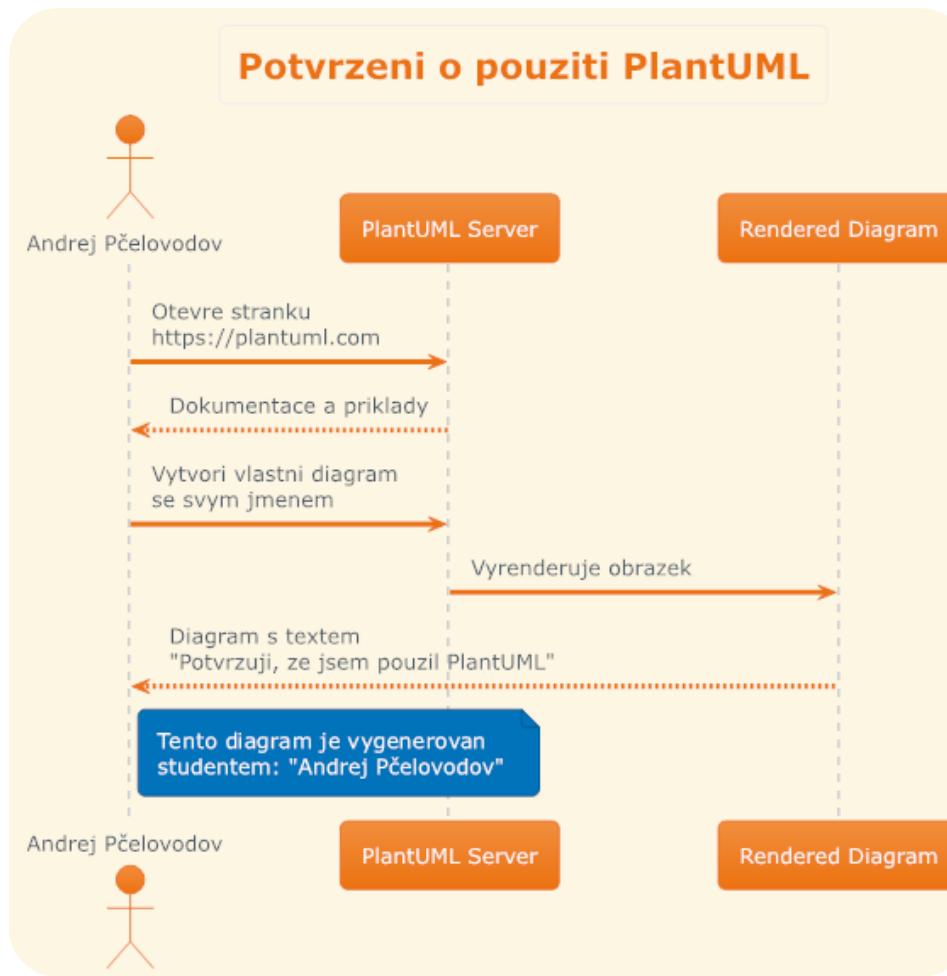
DOMÁCÍ ÚKOL

Nainstalovat:

1. [Eclipse Adoptium JDK 25](#)
2. [Notepad++](#)

Vygenerovat unikátní obrázek se svým jménem.

DOMÁCÍ ÚKOL



Andrej Pčelovodov



POSTUP ODEVZDÁNÍ

1. Otevři záložku Discussions na GitHub
2. Vyber kategorii **03 Homework Submissions**
3. Klikni na zelené tlačítko New Discussion (vpravo nahoře)
4. Název vlákna napiš ve formátu: Homework 03 – Jméno Příjmení



CO MUSÍ ODEVZDÁNÍ OBSAHOVAT

1

PLANTUML KÓD

Vlož svůj kompletní PlantUML kód přímo do příspěvku pomocí code blocku:

```
```
@startuml
Alice -> Bob: Ahoj
Bob --> Alice: Odpověď
@enduml
```
```

Kód musí být celý a spustitelný.



CO MUSÍ ODEVZDÁNÍ OBSAHOVAT

2

VYGENEROVANÝ OBRÁZEK

- Vygeneruj obrázek PNG
- Přetáhni ho přímo do příspěvku (drag & drop)
- Obrázek musí odpovídat vloženému kódu