CONDITIONALS & ITERATION

ŘÍZENÍ TOKU PROGRAMU V PYTHONU

Andrej Pčelovodov

Arvion

CONDITIONALS & ITERATION

ŘÍZENÍ TOKU PROGRAMU V PYTHONU

if • match • for • while • itertools

Andrej Pčelovodov

Arvion

PROČ JE TO DŮLEŽITÉ?

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) ↘️

```python
if user_logged:
    if is_admin:
        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!!!**

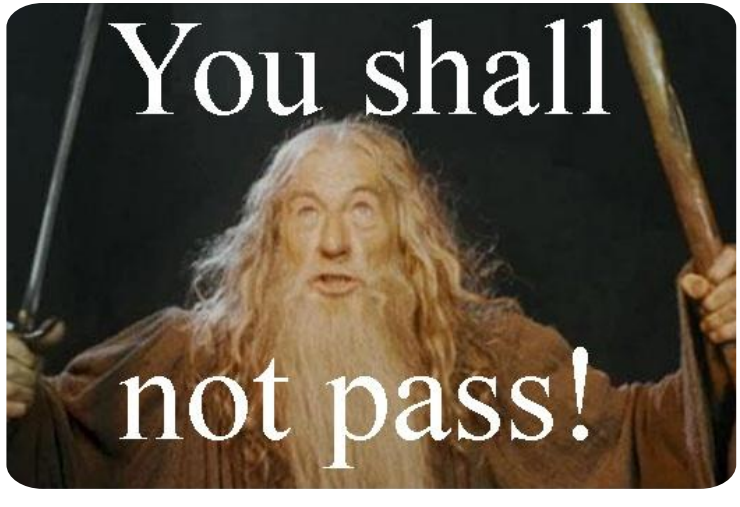
Andrej Pčelovodov

Arvion

VNOŘENÉ PODMÍNKY (NESTED IF) ↘️

```python
if user_logged:
    if is_admin:
        print("Admin panel")
```

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



VNOŘENÉ PODMÍNKY (NESTED IF) ↘️

```python
if user_logged:
    if is_admin:
        print("Admin panel")
```

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

VNOŘENÉ PODMÍNKY (NESTED IF) ↘️

```python
if user_logged:
    if is_admin:
        print("Admin panel")
```

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

```python
if not user_logged and not is_admin:
    return

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

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

```python
n = -5

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)
```

Andrej Pčelovodov

Arvion

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)
```

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)
```

Andrej Pčelovodov

Arvion

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
```

Andrej Pčelovodov

Arvion

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
```

Andrej Pčelovodov

Arvion

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
```

Andrej Pčelovodov

Arvion

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)
```

Andrej Pčelovodov

Arvion

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 počtkrát, 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 počtkrát, 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)
```

Andrej Pčelovodov

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

definicích funkcí

definicích 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

definicích funkcí

definicích 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

definicích funkcí

definicích 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.

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.

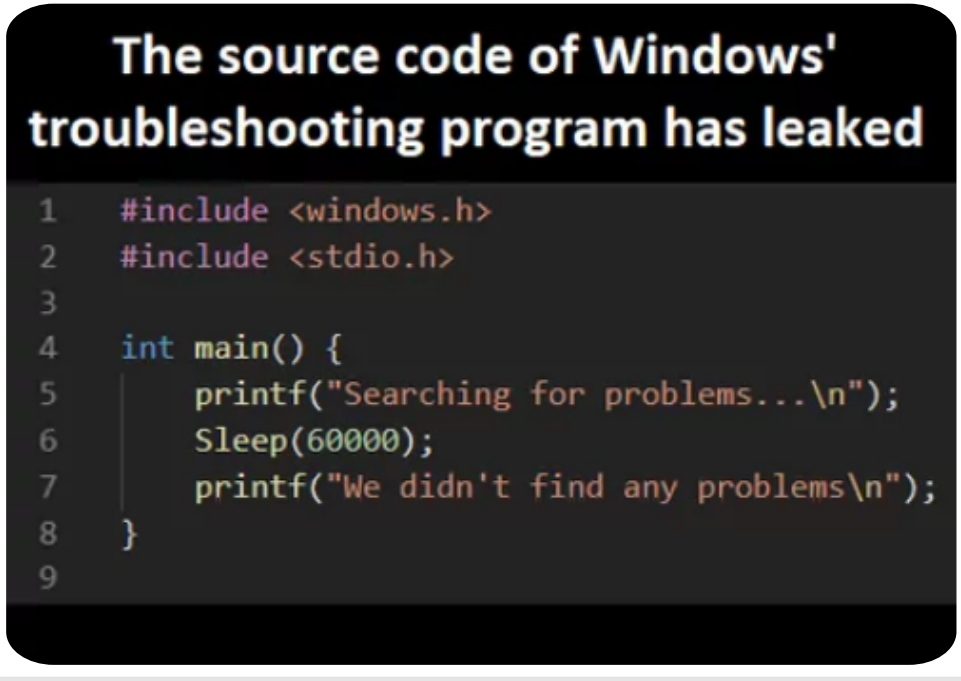
```
1 i = 0
2 while i < 6:
3     i += 1
4     if i == 3:
5         continue
6     print(i)
```

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")
```

PŘÍKLAD Z PRAXE



PŘÍKLAD Z PRAXE

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 :=

WALRUS OPERÁTOR :=

Andrej Pčelovodov

Arvion

WALRUS OPERÁTOR :=

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

WALRUS OPERÁTOR :=

```python
d = [
    {"userId": 1, "name": "rahul", "completed": False},
    {"userId": 1, "name": "rohit", "completed": False},
    {"userId": 1, "name": "ram", "completed": False},
    {"userId": 1, "name": "ravan", "completed": True}
]

print("With Python 3.8 Walrus Operator:")
for entry in d:
    if name := entry.get("name"):
        print(name)

print("Without Walrus operator:")
for entry in d:
    name = entry.get("name")
```

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

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.

```python
from itertools import chain

# a list of odd numbers
odd = [1, 3, 5, 7, 9]

# a list of even numbers
even = [2, 4, 6, 8, 10]

# chaining odd and even numbers
numbers = list(chain(odd, even))

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)))
```

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)
```

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

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