

# Pattern matching v Pythonu

---

- Pavel Tišnovský
  - [kurzy.python@centrum.cz](mailto:kurzy.python@centrum.cz)
- 



---

## Postupné rozšiřování možností Pythonu

- Sémantika
  - (Syntaxe)
- 

## Nová syntaxe+sémantika v Pythonu 3.x

Python 3.5	typové informace
Python 3.6	f-řetězce, async-IO
Python 3.7	klíčová slova async a await
Python 3.8	mroží operátor, poziční parametry
Python 3.9	generické typy
Python 3.10	pattern matching
Python 3.11	skupiny výjimek
Python 3.12	klíčové slovo type + sémantika

---

## Pattern matching

- Přidáno do Pythonu 3.10
  - Zdánlivě lepší varianta konstrukce switch-case
    - ovšem možnosti jsou mnohem větší
  - Další použití
    - zachycení hodnot
    - test typů
    - podmínky v rozhodovacích větvích
    - využití strukturálních vzorů
- 

## Nová stříbrná kulka v IT?

---

### Inspirováno dalšími programovacími jazyky

- SNOBOL
  - AWK
  - ML (Caml, OCaml, F#)
  - Rust
  - Coconut (překládáno do Pythonu)
- 

**SNOBOL**

```

        OUTPUT = "What is your name?"
        Username = INPUT
        Username "J"                                :S(LOVE)
        Username "K"                                :S(HATE)
MEH      OUTPUT = "Hi, " Username                    :(END)
LOVE     OUTPUT = "How nice to meet you, " Username  :(END)
HATE     OUTPUT = "Oh. It's you, " Username
END

```

---

## ML (předchůdce OCamlu a jazyka F#)

```

fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n - 1) + fib (n - 2);

```

---

## ML (předchůdce OCamlu a jazyka F#)

```

fun length(x) = if null(x) then 0
                else 1 + length(tl(x));

```

```

fun length([]) = 0
  | length(a::x) = 1 + length(x)

```

---

## F#

```

let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | n -> fib(n-1) + fib(n-2)

```

---

## Rust

```

fn main() {
    let x:i32 = 1;

    match x {
        0 => println!("zero"),
        1 => println!("one"),
        2 => println!("two"),
        3 => println!("three"),
        _ => println!("something else"),
    }
}

```

---

## Rust

```

fn fib(n: u32) -> u32 {
    match n {
        0 | 1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

fn main() {
    for x in 0..10 {
        println!("{}", x, fib(x))
    }
}

```

### Částečně flexibilní řešení

- Ne všechny vzory je možné použít
  - například "literal" + x + "literal"
  - možná se jejich podpora objeví v další verzi Pythonu?
  - máme příklad implementace: jazyk Coconut

## Ukázky pattern matchingu

### Konstrukce if

```

# Výpočet Ackermannovy funkce, založeno na konstrukci if

def A(m, n):
    """Ackermannova funkce."""
    if m == 0:
        return n + 1
    if n == 0:
        return A(m - 1, 1)
    return A(m - 1, A(m, n - 1))

# otestování korektnosti výpočtu Ackermannovy funkce
for m in range(4):
    for n in range(5):
        print(m, n, A(m, n))

```

### Konstrukce if-else

```

# Výpočet Ackermannovy funkce, založeno na konstrukci if-elif-else

def A(m, n):
    """Ackermannova funkce."""
    if m == 0:
        return n + 1
    elif n == 0:

```

```

        return A(m - 1, 1)
    else:
        return A(m - 1, A(m, n - 1))

# otestování korektnosti výpočtu Ackermannovy funkce
for m in range(4):
    for n in range(5):
        print(m, n, A(m, n))

```

---

## Konstrukce match

```

# Strukturální pattern matching:
# - Výpočet Ackermannovy funkce

def A(m, n):
    """Ackermannova funkce."""
    match (m, n):
        case (0, n):
            return n + 1
        case (m, 0):
            return A(m-1, 1)
        case (m, n):
            return A(m - 1, A(m, n - 1))

# otestování korektnosti výpočtu Ackermannovy funkce
for m in range(4):
    for n in range(5):
        print(m, n, A(m, n))

```

---

```

# Strukturální pattern matching:
# - Výpočet Fibonacciho posloupnost realizovaný s využitím
#   pattern matchingu

def fib(value):
    """Výpočet jednoho prvku Fibonacciho posloupnosti."""
    match value:
        case 0:
            return 0
        case 1:
            return 1
        case n if n>1:
            return fib(n-1) + fib(n-2)
        case _ as wrong:
            raise ValueError("Wrong input", wrong)

# tisk tabulky s prvky Fibonacciho posloupnosti
for n in range(0, 11):

```

```
print(n, fib(n))

# test neplatného vstupu
fib(-1)
```

---

## Rozvětvení

---

```
# Strukturální pattern matching:
# - použití rozhodovací konstrukce if-elif-else
#   namísto pattern matchingu

print("Not ready reading drive A")

def abort_retry_fail():
    # získání odpovědi od uživatele
    response = input("Abort, Retry, Fail? ")

    # rozhodnutí o provedené operaci na základě odpovědi
    if response == "a":
        return "Abort"
    elif response == "r":
        return "Retry"
    elif response == "f":
        return "Fail"
    else:
        return "Wrong response"

print(abort_retry_fail())
```

```
# Strukturální pattern matching:
# - rozhodování realizované slovníkem (mapou)
#   namísto pattern matchingu

print("Not ready reading drive A")

def abort_retry_fail():
    # získání odpovědi od uživatele
    response = input("Abort, Retry, Fail? ")

    # odpovědi a odpovídající operace
    commands = {
        "a": "Abort",
        "r": "Retry",
        "f": "Fail",
    }

    # rozhodnutí o provedené operaci na základě odpovědi
```

```
    return commands.get(response, "Wrong response")
```

```
print(abort_retry_fail())
```

---

```
# Strukturální pattern matching:
```

```
# - rozhodování realizované pattern matchingem
```

```
print("Not ready reading drive A")
```

```
def abort_retry_fail():
```

```
    # získání odpovědi od uživatele
```

```
    response = input("Abort, Retry, Fail? ")
```

```
    # rozhodnutí o provedené operaci na základě odpovědi
```

```
    match response:
```

```
        case "a":
```

```
            return "Abort"
```

```
        case "r":
```

```
            return "Retry"
```

```
        case "f":
```

```
            return "Fail"
```

```
        case _:
```

```
            return "Wrong response"
```

```
print(abort_retry_fail())
```

---

## Python nehlídá, zda jsou pokryty všechny případy

```
# Strukturální pattern matching:
```

```
# - rozhodování realizované pattern matchingem
```

```
print("Not ready reading drive A")
```

```
def abort_retry_fail():
```

```
    # získání odpovědi od uživatele
```

```
    response = input("Abort, Retry, Fail? ")
```

```
    # rozhodnutí o provedené operaci na základě odpovědi
```

```
    match response:
```

```
        case "a":
```

```
            return "Abort"
```

```
        case "r":
```

```
            return "Retry"
```

```
        case "f":
```

```
            return "Fail"
```

```
print(abort_retry_fail())
```

---

## Vzory obsahující v každé větvi větší množství hodnot

---

```
# Strukturální pattern matching:
# - použití rozhodovací konstrukce if-elif-else
# - podpora odpovědí psaných velkými i malými znaky

print("Not ready reading drive A")

def abort_retry_fail():
    # získání odpovědi od uživatele
    response = input("Abort, Retry, Fail? ")

    # rozhodnutí o provedené operaci na základě odpovědi
    if response in {"a", "A", "abort", "Abort", "ABORT"}:
        return "Abort"
    elif response in {"r", "R"}:
        return "Retry"
    elif response in {"f", "F"}:
        return "Fail"
    else:
        return "Wrong response"

print(abort_retry_fail())
```

---

```
# Strukturální pattern matching:
# - rozhodování realizované pattern matchingem
# - podpora odpovědí psaných velkými i malými znaky

print("Not ready reading drive A")

def abort_retry_fail():
    # získání odpovědi od uživatele
    response = input("Abort, Retry, Fail? ")

    # rozhodnutí o provedené operaci na základě odpovědi
    match response:
        case "a" | "A" | "abort" | "Abort" | "ABORT":
            return "Abort"
        case "r" | "R":
            return "Retry"
        case "f" | "F":
            return "Fail"
        case _:
            return "Wrong response"
```

```
print(abort_retry_fail())
```

---

## Zachycení hodnoty proměnné v rozhodovací větvi

---

```
# Strukturální pattern matching:  
# - rozhodování realizované pattern matchingem  
# - podpora odpovědí psaných velkými i malými znaky  
# - zachycení neočekávané odpovědi  
  
print("Not ready reading drive A")  
  
def abort_retry_fail():  
    # získání odpovědi od uživatele  
    response = input("Abort, Retry, Fail? ")  
  
    # rozhodnutí o provedené operaci na základě odpovědi  
    match response:  
        case "a" | "A":  
            return "Abort"  
        case "r" | "R":  
            return "Retry"  
        case "f" | "F":  
            return "Fail"  
        case _ as x:  
            return f"Wrong response {x}"  
  
print(abort_retry_fail())
```

```
# Strukturální pattern matching:  
# - rozhodování realizované pattern matchingem  
  
print("Not ready reading drive A")  
  
def abort_retry_fail():  
    # získání odpovědi od uživatele  
    # a rozhodnutí o provedené operaci  
    match input("Abort, Retry, Fail? "):  
        case "a" | "A":  
            return "Abort"  
        case "r" | "R":  
            return "Retry"  
        case "f" | "F":  
            return "Fail"  
        case _ as x:  
            return f"Wrong response {x}"
```



```
print(abort_retry_fail())
```

---

## Podmínka zapsaná v rozhodovacích větvích konstrukce `match`

- Nazývá se "guard"

---

```
# Strukturální pattern matching:  
# - výpočet faktoriálu s využitím pattern matchingu  
# - základní varianta akceptující neplatné vstupy
```

```
def factorial(n):  
    """Rekurzivní výpočet faktoriálu."""  
    match n:  
        case 0:  
            return 1  
        case 1:  
            return 1  
        case x:  
            return x * factorial(x-1)  
  
# tisk tabulky faktoriálů  
for i in range(0, 10):  
    print(i, factorial(i))
```

---

```
# Strukturální pattern matching:  
# - výpočet faktoriálu s využitím pattern matchingu  
# - test, zda je vstup nezáporný
```

```
def factorial(n):  
    """Rekurzivní výpočet faktoriálu."""  
    match n:  
        case 0:  
            return 1  
        case 1:  
            return 1  
        case x if x>1:  
            return x * factorial(x-1)  
        case _:  
            raise TypeError("expecting integer >= 0")  
  
# tisk tabulky faktoriálů  
for i in range(-1, 10):  
    try:  
        print(i, factorial(i))  
    except Exception as e:  
        print(e)
```

---

```

# Strukturální pattern matching:
# - výpočet faktoriálu s využitím pattern matchingu
# - test, zda má vstup korektní typ

def factorial(n):
    """Rekurzivní výpočet faktoriálu."""
    match n:
        case 0:
            return 1
        case 1:
            return 1
        case x if isinstance(x, int) and x>1:
            return x * factorial(x-1)
        case _:
            raise TypeError("expecting integer >= 0")

# tisk tabulky faktoriálů
for i in range(-1, 10):
    try:
        print(i, factorial(i))
    except Exception as e:
        print(e)

# test reakce na nekorektní vstup
try:
    print(factorial(3.14))
except Exception as e:
    print(e)

# test reakce na nekorektní vstup
try:
    print(factorial("hello"))
except Exception as e:
    print(e)

```

---

```

# Strukturální pattern matching:
# - výpočet faktoriálu s využitím pattern matchingu
# - použití vzoru s operátorem "or"

def factorial(n):
    """Rekurzivní výpočet faktoriálu."""
    match n:
        case 0 | 1:
            return 1
        case x if isinstance(x, int) and x>1:
            return x * factorial(x-1)
        case _:
            raise TypeError("expecting integer >= 0")

```

```


# tisk tabulky faktoriálů
for i in range(-1, 10):
    try:
        print(i, factorial(i))
    except Exception as e:
        print(e)

# test reakce na nekorektní vstup
try:
    print(factorial(3.14))
except Exception as e:
    print(e)

# test reakce na nekorektní vstup
try:
    print(factorial("hello"))
except Exception as e:
    print(e)

```

## Pattern matching a výčtový typ

 red blue pill

```

from enum import Enum

class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

match pill:
    case Color.RED:
        print("you stay in Wonderland, and I show you how deep the rabbit hole goes.")
    case Color.BLUE:
        print("the story ends, you wake up in your bed and believe whatever you want to believe")
    case _:
        print("this does not compute")

```

## Pattern matching a n-tice a seznamy

```

[x, y, *rest]
(x, y, *rest)
(x, y, *_)

```

```

# Strukturální pattern matching:
# - komplexní čísla realizovaná formou dvojice hodnot

```

```
def test_number(value):
    """Test, o jakou variantu komplexního čísla se jedná."""
    match value:
        case (0, 0):
            print("Zero")
        case (real, 0):
            print(f"Real number {real}")
        case (0, imag):
            print(f"Imaginary number {imag}")
        case (real, imag):
            print(f"Complex number {real}+i{imag}")
        case _:
            raise ValueError("Not a complex number")

test_number((0,0))
test_number((1,0))
test_number((0,1))
test_number((1,1))
```

---

## Seznamy, n-tice a podmínky pro hodnoty prvků těchto kolekcí

---

```
# Strukturální pattern matching:
# - komplexní čísla realizovaná formou dvojice hodnot

def test_number(value):
    """Test, o jakou variantu komplexního čísla se jedná."""
    match value:
        case (0, 0):
            print("Zero")
        case (real, 0) if real>0:
            print(f"Positive real number {real}")
        case (real, 0):
            print(f"Negative real number {real}")
        case (0, imag) if imag<0:
            print(f"Negative imaginary number {imag}")
        case (0, imag):
            print(f"Positive imaginary number {imag}")
        case (real, imag):
            print(f"Complex number {real}+i{imag}")
        case _:
            raise ValueError("Not a complex number")

test_number((0,0))
test_number((1,0))
test_number((-1,0))
test_number((0,1))
test_number((0,-1))
test_number((1,1))
```

---

```
# Strukturální pattern matching:
# - komplexní čísla realizovaná formou seznamu dvou hodnot

def test_number(value):
    """Test, o jakou variantu komplexního čísla se jedná."""
    match value:
        case [0, 0]:
            print("Zero")
        case [real, 0] if real>0:
            print(f"Positive real number {real}")
        case [real, 0]:
            print(f"Negative real number {real}")
        case [0, imag] if imag<0:
            print(f"Negative imaginary number {imag}")
        case [0, imag]:
            print(f"Positive imaginary number {imag}")
        case [real, imag]:
            print(f"Complex number {real}+i{imag}")
        case _:
            raise ValueError("Not a complex number")

test_number([0,0])
test_number([1,0])
test_number([-1,0])
test_number([0,1])
test_number([0,-1])
test_number([1,1])
```

---

## Zpracování příkazů či strukturovaných textových souborů s využitím pattern matchingu

---

```
# Strukturální pattern matching:
# - rozpoznání a zpracování příkazů zadaných uživatelem
# - nejjednodušší podoba pro víceslovní příkazy

def perform_command():
    # získat příkaz od uživatele
    response = input("> ")

    match response:
        case "quit":
            return "Quit"
        case "list employees":
            return "List employees"
        case "list departments":
            return "List departments"
        case "list rooms":
            return "List rooms"
        case _:
```

```
        return "Wrong command"
```

```
print(perform_command())
```

---

## Rozdělení na jednotlivá slova

---

```
# Strukturální pattern matching:  
# - rozpoznání a zpracování příkazů zadaných uživatelem  
# - rozdělení příkazů na jednotlivá slova
```

```
def perform_command():  
    # získat příkaz od uživatele  
    response = input("> ")  
  
    match response.split():  
        case ["quit"]:  
            return "Quit"  
        case ["list", "employees"]:  
            return "List employees"  
        case ["list", "departments"]:  
            return "List departments"  
        case ["list", "rooms"]:  
            return "List rooms"  
        case _:  
            return "Wrong command"
```

```
print(perform_command())
```

---

## Rozpoznání a zpracování proměnné části víceslovních příkazů

---

```
# Strukturální pattern matching:  
# - rozpoznání a zpracování příkazů zadaných uživatelem  
# - zachycení a zpracování nekonstantního (proměnného) slova
```

```
def perform_command():  
    # získat příkaz od uživatele  
    response = input("> ")  
  
    match response.split():  
        case ["quit"]:  
            return "Quit"  
        case ["list", "employees"]:  
            return "List employees"  
        case ["list", "departments"]:  
            return "List departments"  
        case ["list", "rooms"]:  
            return "List rooms"  
        case ["info", subject]:
```

```
        return f"Info about subject '{subject}'"
    case _:
        return "Wrong command"

print(perform_command())
```

---

## Vnořené řídicí struktury `match`

---

```
# Strukturální pattern matching:
# - rozpoznání a zpracování příkazů zadaných uživatelem
# - ukázka použití vnořených řídicích struktur match.

def perform_command():
    # získat příkaz od uživatele
    response = input("> ")

    # rozvětvení na základě prvního slova
    match response.split():
        case ["quit"]:
            return "Quit"
        case ["list", obj]:
            # rozvětvení na základě druhého slova
            match obj:
                case "employees":
                    return "List employees"
                case "departments":
                    return "List departments"
                case "rooms":
                    return "List rooms"
                case _:
                    return "Invalid object: employees, departments, or rooms expected"
        case ["info", subject]:
            return f"Info about subject '{subject}'"
        case _:
            return "Wrong command"

print(perform_command())
```

---

```
# Strukturální pattern matching:
# - rozpoznání a zpracování příkazů zadaných uživatelem
# - ukázka použití vnořených řídicích struktur match
# - omezení hodnoty druhého slova v příkazu "list"

def perform_command():
    # získat příkaz od uživatele
    response = input("> ")

    # rozvětvení na základě prvního slova
```

```

match response.split():
    case ["quit"]:
        return "Quit"
    case ["list", ("employees" | "departments" | "rooms") as obj]:
        # rozvětvení na základě druhého slova
        match obj:
            case "employees":
                return "List employees"
            case "departments":
                return "List departments"
            case "rooms":
                return "List rooms"
    case ["info", subject]:
        return f"Info about subject '{subject}'"
    case _:
        return "Wrong command"

print(perform_command())

```

---

## Zachycení dopředu neznámého počtu hodnot

---

```

# Strukturální pattern matching:
# - rozpoznání víceslovních příkazů

def perform_command():
    response = input("> ")

    match response.split():
        case ["quit"]:
            return "Quit"
        case ["list", "employees"]:
            return "List employees"
        case ["list", "departments"]:
            return "List departments"
        case ["list", "rooms"]:
            return "List rooms"
        case ["info", *subjects]:
            return f"Info about {len(subjects)} subjects '{subjects}'"
        case _:
            return "Wrong command"

print(perform_command())

```

```

# Strukturální pattern matching:
# - rozpoznání víceslovních příkazů

```



```
def perform_command():
    response = input("> ")

    match response.split():
        case ["quit"]:
            return "Quit"
        case ["list", "employees"]:
            return "List employees"
        case ["list", "departments"]:
            return "List departments"
        case ["list", "rooms"]:
            return "List rooms"
        case ["info", *subjects] if len(subjects) > 0:
            return f"Info about {len(subjects)} subjects '{subjects}'"
        case _:
            return "Wrong command"

print(perform_command())
```

---

## Strukturální pattern matching a objekty

---

```
# Strukturální pattern matching:
# - pattern matching a objekty

class Complex():
    """Třída představující komplexní čísla."""

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        return f"Complex number {self.real}+i{self.imag} represented as object"

def test_number(value):
    """Test, o jakou variantu komplexního čísla se jedná."""
    match value:
        case (0, 0):
            print("Zero")
        case (real, 0) if real>0:
            print(f"Positive real number {real}")
        case (real, 0):
            print(f"Negative real number {real}")
        case (0, imag) if imag<0:
            print(f"Negative imaginary number {imag}")
        case (0, imag):
            print(f"Positive imaginary number {imag}")
        case (real, imag):
            print(f"Complex number {real}+i{imag}")
```

```

    case Complex():
        print(value)
    case _:
        raise ValueError("Not a complex number")

```

```

test_number((0,0))
test_number((1,0))
test_number((-1,0))
test_number((0,1))
test_number((0,-1))
test_number((1,1))

```

```

test_number(Complex(0,0))
test_number(Complex(1,0))
test_number(Complex(-1,0))
test_number(Complex(0,1))
test_number(Complex(0,-1))
test_number(Complex(1,1))

```

---

```

# Strukturální pattern matching:
# - pattern matching a objekty

```

```

from fractions import Fraction

```

```

class Complex():
    """Třída představující komplexní čísla."""

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        return f"Complex number {self.real}+i{self.imag} represented as object"

```

```

def test_number(value):
    """Test, o jakou variantu komplexního čísla se jedná."""
    match value:
        case (0, 0):
            print("Zero")
        case (real, 0) if real>0:
            print(f"Positive real number {real}")
        case (real, 0):
            print(f"Negative real number {real}")
        case (0, imag) if imag<0:
            print(f"Negative imaginary number {imag}")
        case (0, imag):
            print(f"Positive imaginary number {imag}")
        case (real, imag):

```

```

        print(f"Complex number {real}+i{imag}")
    case Complex(real=0, imag=0):
        print("Zero complex represented as object")
    case Complex():
        print(value)
    case Fraction():
        print(f"Fraction {value}")
    case _:
        raise ValueError("Not a complex number")

test_number((0,0))
test_number((1,0))
test_number((-1,0))
test_number((0,1))
test_number((0,-1))
test_number((1,1))

test_number(Complex(0,0))
test_number(Complex(1,0))
test_number(Complex(-1,0))
test_number(Complex(0,1))
test_number(Complex(0,-1))
test_number(Complex(1,1))

test_number(Fraction(0,1))
test_number(Fraction(1,1))
test_number(Fraction(1,2))
test_number(Fraction(1,3))

```

---

## Rozpoznání typu výjimky

```

# Strukturální pattern matching:
# - reakce na různé typy výjimek

def parse_generic_llm_error(e: Exception) -> tuple[int, str, str]:
    """Try to parse generic LLM error."""
    match e:
        case BadRequestError():
            return parse_openai_error(e)
        case ApiResponseException():
            return parse_bam_error(e)
        case ApiRequestFailure():
            return parse_watsonx_error(e)
        case _:
            return DEFAULT_STATUS_CODE, DEFAULT_ERROR_MESSAGE, str(e)

```

---

## Ucelený příklad na konec: reprezentace barev různými metodami

```

# Praktická ukázka pattern matchingu:
# - převod barvy z různých reprezentací do barvového prostoru RGB

```

```

from dataclasses import dataclass
from enum import Enum

class BasicColor(Enum):
    """Základních osm barev."""
    BLACK = (0, 0, 0)
    RED = (255, 0, 0)
    GREEN = (0, 255, 0)
    YELLOW = (255, 255, 0)
    BLUE = (0, 0, 255)
    MAGENTA = (255, 0, 255)
    CYAN = (0, 255, 255)
    WHITE = (255, 255, 255)

@dataclass
class Gray:
    """Reprezentace odstínu šedi celočíselnou hodnotou 0..255."""
    gray : int

@dataclass
class RGB:
    """Reprezentace barvy v barvovém prostoru RGB."""
    red : int
    green : int
    blue : int

@dataclass
class HSV:
    """Reprezentace barvy v barvovém prostoru HSV."""
    hue : float
    saturation: float
    value : float

def scale_rgb(r, g, b):
    """Převod hodnot z rozsahu 0.0-1.0 na celočíselný rozsah 0..255."""
    return RGB(int(255*r), int(255*g), int(255*b))

def hsv_to_rgb(hue, saturation, value):
    """Převod barvy z barvového prostoru HSV do prostoru RGB."""
    if saturation==0:
        return scale_rgb(value, value, value)
    else:
        return hsv_to_rgb_(hue, saturation, value)

```

```

def hsv_to_rgb(hue, saturation, value):
    """Pomocná funkce pro výpočet hodnot RGB."""
    if hue == 1.0:
        hue = 0.0
    i = int(hue*6.0)
    f = hue*6.0 - i

    w = value * (1.0 - saturation)
    q = value * (1.0 - saturation * f)
    t = value * (1.0 - saturation * (1.0 - f))

    match i:
        case 0:
            return scale_rgb(value, t, w)
        case 1:
            return scale_rgb(q, value, w)
        case 2:
            return scale_rgb(w, value, t)
        case 3:
            return scale_rgb(w, q, value)
        case 4:
            return scale_rgb(t, w, value)
        case 5:
            return scale_rgb(value, w, q)

def to_rgb(color):
    """Převod barvy z jakékoli podporované reprezentace do prostoru RGB."""
    match color:
        case Gray(gray):
            return RGB(gray, gray, gray)
        case RGB() as rgb:
            return rgb
        case HSV(hue, saturation, value):
            return hsv_to_rgb(hue, saturation, value)
        case BasicColor() as b:
            return RGB(*b.value)
        case _:
            return f"Invalid color {color}"

# otestování jednotlivých možností

print("Grayscale:")

gray_color1 = Gray(0)
print(to_rgb(gray_color1))

gray_color2 = Gray(255)
print(to_rgb(gray_color2))

print("\nRGB:")

```

```
rgb_color1 = RGB(0, 0, 0)
print(to_rgb(rgb_color1))

rgb_color2 = RGB(0, 255, 0)
print(to_rgb(rgb_color2))

rgb_color3 = RGB(255, 255, 255)
print(to_rgb(rgb_color3))

print("\nHSV:")

hsv_color1 = HSV(0.0, 0.0, 1.0)
print(to_rgb(hsv_color1))

hsv_color2 = HSV(0.0, 0.0, 0.5)
print(to_rgb(hsv_color2))

hsv_color3 = HSV(0.0, 1.0, 1.0)
print(to_rgb(hsv_color3))

hsv_color4 = HSV(0.3333, 1.0, 1.0)
print(to_rgb(hsv_color4))

hsv_color5 = HSV(0.6666, 1.0, 1.0)
print(to_rgb(hsv_color5))

hsv_color6 = HSV(1.0, 1.0, 1.0)
print(to_rgb(hsv_color6))

hsv_color7 = HSV(1.0, 0.5, 0.5)
print(to_rgb(hsv_color7))

print("\nBasic colors:")

basic_color1 = BasicColor.BLACK
print(to_rgb(basic_color1))

basic_color2 = BasicColor.RED
print(to_rgb(basic_color2))

basic_color3 = BasicColor.GREEN
print(to_rgb(basic_color3))

basic_color4 = BasicColor.BLUE
print(to_rgb(basic_color4))

basic_color5 = BasicColor.YELLOW
print(to_rgb(basic_color5))

basic_color6 = BasicColor.MAGENTA
print(to_rgb(basic_color6))
```

```
basic_color7 = BasicColor.CYAN
print(to_rgb(basic_color7))
```

```
basic_color8 = BasicColor.WHITE
print(to_rgb(basic_color8))
```

---

```
def exit():
    .0%0.
```