2. Řídicí struktury

Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Řízení toku výpočtu
  - Logické výrazy
  - Větvení
  - Cykly
- Část 2 – Funkce
- Část 3 – Ladění

Část I

Řízení toku výpočtu

I. Řízení toku výpočtu

Logické výrazy

Větvení

Cykly

Porovnávání (čísel)

- Výsledkem operace porovnávání je logická hodnota **True** nebo **False** (datový typ `bool`)
- Operátory `>`, `<`, `==`, `>=`, `<=`, `!=`

```
>>> 8 > 3
True
>>> 10 <= 10
True
>>> 1==0
False
>>> 2!=3
True
>>> a=4
>>> b=6
>>> a<b
True
```

Logické operátory

- Python má tři logické opeátory: `and`, `or` a `not`
- Nejvyšší prioritu má `not`, následován `and`, nejnižší má `or`

```
>>> not True
False
>>> not False
True
```

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
```

Exkluzivní logický součet – XOR

- Řekněme, že máte dvě logické proměnné, `b1` a `b2`, a chcete výraz, který se vyhodnotí jako `True` tehdy, když právě jedna z hodnot `b1` nebo `b2` je `True`.
- Výraz `b1 and non b2` nabývá hodnoty `True`, pokud je `b1 = True` a `b2 = False`

```
>>> b1 = True; b2 = False
>>> (b1 and not b2)
True
```

- Výraz `b2 and non b1` nabývá hodnoty `True`, pokud je `b2 = True` a `b1 = False`.

```
>>> b1 = False; b2 = True
>>> (b2 and not b1)
True
```

- Nemůže se stát, aby oba tyto současně nabývaly hodnoty `True`.
- Pokud jsou `b1` i `b2` současně `True` nebo `False`, vyhodnotí se oba výrazy jako `False`.
- Můžeme tedy oba výrazy spojit pomocí `or`. Vyzkoušejte! Je možný kratší zápis?

I. Řízení toku výpočtu

Logické výrazy

Větvení

Cykly

Podmíněný příkaz `if`

- Umožňuje větvení programu na základě podmínky
- Má tvar:

```
if podminka:
  prikaz1
else:
  prikaz2
```

- `podminka` – logický výraz, jehož hodnota je logického typu

  False (hodnota 0) nebo True (hodnota různá od 0)

- podle toho, jak se provede podmínka, se provede jedna z větví
- `else` větev nepovinná
- možné vícenásobné větvení

Podmíněný příkaz `if`

```
# Demonstrace podmineneho prikazu if
import sys

n = int(sys.argv[1])
# první argument - cele číslo
if n>0:
    print(n, " je kladne cislo.")

print("Konec programu.")
```

lec02/conditionals.py

```
$ python conditionals.py 10
10 je kladne cislo.
Konec programu.
$ python conditionals.py -1
Konec programu.
```

Podmíněný příkaz `if`

```
# Demonstrace podmineneho prikazu if
import sys

n = int(sys.argv[1])
# první argument - cele číslo
if n>0:
    print(n, " je kladne cislo.")

print("Konec programu.")
```

lec02/conditionals.py

```
$ python conditionals.py 10
10 je kladne cislo.
Konec programu.
$ python conditionals.py -1
Konec programu.
```

Bloky kódu

- v Pythonu určené odsazením
- základ strukturovaného programování

Větvení `if-else`

```python
# Demonstrace vetveni if-else
import sys

n = int(sys.argv[1])
# první argument
if n>0:
    print(n, " je kladne cislo.")
else:
    print(n, " neni kladne cislo.")
```

lec02/conditionals2.py

```
$ python conditionals2.py 10
10 je kladne cislo.
$ python conditionals.py -5
-5 neni kladne cislo.
```

Vnořené větvení

```python
# Demonstrace vnoreneho vetveni
import sys

n = int(sys.argv[1])
# první argument
if n>0:
    print(n, " je kladne cislo")
else:
    if n==0:
        print(n, " je nula")
    else:
        print(n, " je zaporne cislo")
```

lec02/conditionals3.py

```
$ python conditionals3.py 0
0 je nula.
```

Zřetězené podmínky `if-elif-else`

```
# Demonstrace zretezeneho vetveni
import sys

n = int(sys.argv[1])
# prvni argument
if n>0:
    print(n, " je kladne cislo")
elif n==0:
    print(n, " je nula")
else:
    print(n, " je zaporne cislo")
```

lec02/conditionals4.py

```
$ python conditionals4.py 0
0 je nula.
```

Příklad – maximum tří čísel

```
# Program vytiskne maximum tri zadanych cisel
import sys

a=int(sys.argv[1]); b=int(sys.argv[2]); c=int(sys.argv[3])

if a>b:    # a nebo c
  if a>c: # a > b, a > c
    print(a)
  else:    # c >= a > b
    print(c)
else:      # b >= a
  if b>c: # b > c, b >= a
    print(b)
  else:    # c >= b >= a
    print(c)
```

lec02/maxi.py

Ternární operátor

- Jednořádkový zápis jednoduchého větvení

```
>>> is_student = True
>>> person = 'student' if is_student else 'not student'
>>> print(person)
student
```

- Odpovídá konstrukci

```
>>> if is_student:
...     person = 'student'
... else:
...     person = 'not student'
...
>>> print(person)
student
```

I. Řízení toku výpočtu

Logické výrazy

Větvení

Cykly

Cyklus `for`, `range`

- Známý počet opakování cyklu:
- Má tvar:

```
for x in range(n):
  prikaz
```

- provede příkazy pro všechny hodnoty `x` ze zadaného intervalu
- `range(n)` – interval od `0` do `n-1` (tj. `n` opakování)
- `range(a, b)` – interval od `a` do `b-1`
- `for`/`range` lze použít i obecněji (nejen intervaly) – viz později/samostudium

Příklady

```python
# Program vypocita faktorial zadaneho cisla
n = int(input())
# proměnna bude pouzita ve vypoctu, musi byt deklarovana
# pozor na to, jakou bude mit hodnotu
f = 1
# range vraci rozsah 0 - n-1
for i in range(1, n+1):
    f = f * i   # lze zkratit: f *= i
# tisk vysledku
print(f)
```

lec02/factorial-for.py

Vnořené cykly

- Řídicí struktury můžeme zanořovat, např.:
  - podmínka uvnitř cyklu
  - cyklus uvnitř cyklu

Počet zanoření je neomezený, ale...

```
>>> n = 5

>>> for i in range(1, n**2 + 1):
...     print(i, end=" ")
...     if not i % n:
...         print()
...
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

lec02/table.py

Vnořené cykly

- Řídicí struktury můžeme zanořovat, např.:
  - podmínka uvnitř cyklu
  - cyklus uvnitř cyklu

Počet zanoření je neomezený, ale...

```
>>> n = 5

>>> for i in range(1, n**2 + 1):
...     print(i, end=" ")
...     if not i % n:
...         print()
...
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

lec02/table.py

Příklad – hezčí formátování

```
>>> n = 8

>>> for i in range(1, n+1):
...     for j in range(n):
...         print(str(i+j).ljust(2), end=" ")
...     print()
...
1  2  3  4  5  6  7  8
2  3  4  5  6  7  8  9
3  4  5  6  7  8  9  10
4  5  6  7  8  9  10 11
5  6  7  8  9  10 11 12
6  7  8  9  10 11 12 13
7  8  9  10 11 12 13 14
8  9  10 11 12 13 14 15
```

lec02/table2.py

Cyklus `while`

- Neznámý počet opakování, provádí příkazy dokud platí podmínka
- Má tvar:

```
while podminka:
  prikaz1
else:
  prikaz2
```

- V podmínce je vyhodnocována hodnota řidící proměnné, která se mění v těle cyklu
- Část `else` není povinná
- Může se stát:
  - neprovede příkazy ani jednou
  - provádí příkazy do nekonečna (nikdy neskončí)

To většinou znamená chybu v programu

Příklad

```
# Program factorial-while.py
# Vypocita faktorial zadaneho cisla
n = int(input())

f = 1

while n > 0:
    f = f * n
    n = n - 1

print(f)
```

lec02/factorial-while.py

Přerušení cyklu – příkazy `break` a `continue`

- `break` přeruší cyklus

```
>>> for i in range(5):
...     if i==3:
...         break
...     print(i, end='')
...
012
```

- `continue` přeruší aktuální iteraci a začne následující

```
>>> for i in range(5):
...     if i==3:
...         continue
...     print(i, end='')
...
0124
```

Příklad – binární zápis čísla

```
>>> n = 45
>>> output = ""

>>> while n > 0:
...     if n % 2 == 0:
...         output = "0" + output
...     else:
...         output = "1" + output
...     n = n // 2 # integer division
...
>>> print(output)
101101
```

lec02/dec2bin.py

Část II

Funkce

Strukturovaný kód

Programy nepíšeme jako jeden dlouhý štrůdl, ale chceme je strukturovat.

Proč?

- opakované provádění stejného (velmi podobného) kódu na různých místech algoritmu
- modularita (viz Lego kostky), znovupoužitelnost
- snažší uvažování o problému, čitelnost, dělba práce

Prostředky pro strukturování kódu

- Bloky kódu – *(oddělené odsazením)*, např:
- Funkce
- Moduly, programy (soubory)

Příklad existujících funkcí – knihovna `math`

- použití knihovny: `import math`
- zaokrouhlování: `round`, `math.ceil`, `math.floor`
- absolutní hodnota: `abs`
- `math.exp`, `math.log`, `math.sqrt`
- goniometrické funkce: `math.sin`, `math.cos`, ...
- konstanty: `math.pi`, `math.e`

```
>>> import math
>>> math.sqrt(4.0)
2.0
>>> math.sin(30./180.*math.pi)
0.49999999999999994
>>> math.exp(1.0)
2.718281828459045
```

Definice uživatelských funkcí

- Příklad – druhá mocnina

```
>>> def square(x):
...     return x*x
...
>>> square(3)
9
```

- Obecně:

```
def jmeno ( parametry ):
  <blok kodu>
```

- `return(value)` – návrat do nadřazené funkce
- problémy bychom měli členit na podproblémy – zde např. můžeme funkci využít pro výpočet přepony

Příklad uživatelské funkce – binární zápis čísla

```
>>> def to_binary(n):
...     output = ""
...     while n > 0:
...         if n % 2 == 0:
...             output = "0" + output
...         else:
...             output = "1" + output
...         n = n // 2
...     return output
...
>>> print(to_binary(22))
10110
>>> print(to_binary(16))
10000
```

lec02/dec2bin-func.py

Vlastnosti funkcí

- **vstup:** parametry funkce
- **výstup:** návratová hodnota (předaná zpět pomocí `return`)
  - `return` není `print`
  - upozornění: `yield` – podobné jako `return`, pokročilý konstrukt, v tomto kurzu nebudeme nepoužívat
- proměnné v rámci funkce:
  - lokální: dosažitelné pouze v rámci funkce
  - globální: dosažitelné všude, minimalizovat použití (více později)
- funkce mohou volat další funkce
  - po dokončení vnořené funkce se interpret vrací a pokračuje
- **rekurze:** volání sebe sama, cyklické volání funkcí (podrobněji později – 6. přednáška)

Příklad – vnořené volání funkcí

```
>>> def parity_info(number):
...     print("Číslo je", number, end=" ")
...     if number % 2 == 0:
...         print("sudé")
...     else:
...         print("liché")
...
>>> def parity_experiment(a, b):
...     parity_info(a)
...     parity_info(b)
...

>>> parity_experiment(3, 18)
Číslo je 3 liché
Číslo je 18 sudé
```

lec02/nested-function.py

Funkce – speciality Pythonu

```
>>> def test(x, y=3):
...     print("X =", x, ", Y =", y)
...
```

- defaultní hodnoty parametrů
- volání pomocí jmen parametrů
- funkci test lze volat např.

```
>>> test(2, 8)
X = 2 , Y = 8
>>> test(1)
X = 1 , Y = 3
>>> test(y=5, x=4)
X = 4 , Y = 5
```

- (dále též libovolný počet parametrů a další speciality)

Návrh funkcí

- Specifikace: vstupně-výstupní chování
  - ujasnit si před psaním samotného kódu
  - jaké potřebuje funkce vstupy?
  - co je výstupem funkce
- Funkce by měly být krátké:
  - jedna myšlenka
  - max na jednu obrazovku
  - jen pár úrovní zanoření
- Příliš dlouhá funkce – rozdělit na kratší
- Funkce **vrací hodnotu** vypočítanou ze **vstupních argumentů**.
- **Čistá funkce** (pure function) – výstup závisí pouze na vstupních parametrech, a to jednoznačně.

Příklad – tisk šachovnice

```
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
```

1. řešení
   - Dvě různé funkce pro tisk sudých a lichých řádků
   - Parametrem funkce délka řádku
   - V nadřízené funkci je vyhodnocena sudost nebo lichost řádku a zavolána příslušná funkce
2. řešení
   - Funkce pro kreslení jenom jedna
   - Dalším parametrem je parita
3. řešení
   - Jedna funkce
   - Vyhodnocuje se parita pořadí znaku
   - Ve vhodnou chvíli se odřádkuje

Příklad – tisk šachovnice

```
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
```

1. řešení
   - Dvě různé funkce pro tisk sudých a lichých řádků
   - Parametrem funkce délka řádku
   - V nadřízené funkci je vyhodnocena sudost nebo lichost řádku a zavolána příslušná funkce
2. řešení
   - Funkce pro kreslení jenom jedna
   - Dalším parametrem je parita
3. řešení
   - Jedna funkce
   - Vyhodnocuje se parita pořadí znaku
   - Ve vhodnou chvíli se odřádkuje

Příklad – tisk šachovnice

```
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#.#.#.
.#.#.#.#
```

1. řešení
   - Dvě různé funkce pro tisk sudých a lichých řádků
   - Parametrem funkce délka řádku
   - V nadřízené funkci je vyhodnocena sudost nebo lichost řádku a zavolána příslušná funkce
2. řešení
   - Funkce pro kreslení jenom jedna
   - Dalším parametrem je parita
3. řešení
   - Jedna funkce
   - Vyhodnocuje se parita pořadí znaku
   - Ve vhodnou chvíli se odřádkuje

Tisk šachovnice – 1. řešení

```
>>> def even_line(n):
...     for j in range(n):
...         if (j % 2 == 0):
...             print("#", end="")
...         else:
...             print(".", end="")
...     print()
...
```

```
>>> def odd_line(n):
...     for j in range(n):
...         if (j % 2 == 1):
...             print("#", end="")
...         else:
...             print(".", end="")
...     print()
...
```

```
>>> def chessboard(n):
...     for i in range(n):
...         if (i % 2 == 0):
...             even_line(n)
...         else:
...             odd_line(n)
...
```

```
>>> chessboard(5)
#.#.#
.#.#.
#.#.#
.#.#.
#.#.#
```

lec02/chessboard1.py

Tisk šachovnice – 1. řešení

```
>>> def even_line(n):
...     for j in range(n):
...         if (j % 2 == 0):
...             print("#", end="")
...         else:
...             print(".", end="")
...     print()
...
>>> def chessboard(n):
...     for i in range(n):
...         if (i % 2 == 0):
...             even_line(n)
...         else:
...             odd_line(n)
...
```

```
>>> def odd_line(n):
...     for j in range(n):
...         if (j % 2 == 1):
...             print("#", end="")
...         else:
...             print(".", end="")
...     print()
...
>>> chessboard(5)
#.#.#
.#.#.
#.#.#
.#.#.
#.#.#
```

lec02/chessboard1.py

Tisk šachovnice – 2. řešení

```
>>> def chessboard(n):
...     for i in range(n):
...         line(n, i % 2)
...
>>> def line(n, parity):
...     for j in range(n):
...         if (j % 2 == parity):
...             print("#", end="")
...         else:
...             print(".", end="")
...     print()
...
```

lec02/chessboard2.py

Tisk šachovnice – 3. řešení

```
>>> def chessboard(n):
...     for i in range(n):
...         for j in range(n):
...             c = "#" if ((i+j) % 2 == 0) else "."
...             print(c, end="")
...         print()
...
>>> chessboard(5)
#.#.#
.#.#.
#.#.#
.#.#.
#.#.#
```

lec02/chessboard3.py

Počet parametrů

```
def funkce(a, b, c):
  <blok kodu>
```

- Počet parametrů je dán hlavičkou funkce
- Existují i funkce bez parametrů

Nemůže být čistá, pokud není konstatní.

- Parametry mohou mít defaultní hodnotu

Existují i funkce s proměnným počtem parametrů.

Návratová hodnota

Příkazů `return` může být ve funkci několik

```
>>> def isprime(n):
...     p=2
...     while p*p<=n:
...         if n % p == 0:
...             return False
...         p+=1
...     return True
...
>>> isprime(16)
False
>>> isprime(17)
True
```

Funkce nemusí vracet nic, pak často hovoříme o proceduře nebo void funkci. V proceduře můžeme explicitně specifikovat **return None**

Více návratových hodnot

```
>>> def f(x):
...     return x, 2*x, 3*x
...
>>> a, b, c = f(10)
>>> a
10
>>> b
20
>>> c
30
```

- `(x,2*x,3*x)` je objekt typu `tuple` (uspořádaná n-tice)
- Této technice se říká rozbalení (unpacking)

Návratová hodnota None

- Funkce bez return vrací hodnotu None

```
>>> def f(x):
...     x = x*2
...
>>> res = f(2)
>>> print(res)
None
```

- None se používá k explicitnímu vyjádření, že funkce nevrací žádnou hodnotu

```
>>> def f(x):
...     x = x*2
...     return None
...
```

Rozsah platnosti proměnných

- Proměnné definované
  - v hlavním programu (mimo funkce) jsou **globální**, viditelné všude
  - uvnitř funkce jsou **lokální**, viditelné pouze uvnitř této funkce

```
>>> k=3 # je globalni
>>> def f(x):
...     l=2*x     # a je lokalni
...     print(k,l)
...
>>> f(1)
3 2
>>> print(k)
3
>>> print(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'l' is not defined
```

Funkce jako argument

```
>>> def twice(f,x):
...     return f(f(x))
...
>>> def square(x):
...     return x*x
...
>>> print(twice(square,10))
10000
```

```
>>> def repeatNtimes(f,n):
...     for i in range(n): f()
...
>>> def ahoj():
...     print("Ahoj")
...
>>> repeatNtimes(ahoj,4)
Ahoj
Ahoj
Ahoj
Ahoj
```

Část III

Chyby a ladění

Poznámka o ladění

- Laděním se nebudeme (na přednáškách) příliš zabývat
- ... to ale neznamená, že není důležité !

> Ladění je dvakrát tak náročné, jak psaní vlastního kódu. Takže pokud napíšete program tak chytře, jak jen umíte, nebudete schopni jej odladit. (Brian W. Kernighan)

Jak na to?

- Ladící výpisy
  - např. v každé iteraci cyklu vypisujeme stav proměnných
  - doporučeno vyzkoušet na ukázkových programech z přednášek
- Použití debuggeru
  - bývá dostupný přímo v IDE (IDLE, ...)
  - sledování hodnot proměnných, spuštěných příkazů, breakpointy, ...
- Komnpozice na funkce
  - chyba se daleko lépe hledá v dílčí funkci než v celém programu najednou

Základní typy chyb

- Syntaktické chyby – špatný zápis programu, odhalí statická analýza
  - **SyntaxError** – zapomenutá dvojtečka či závorka, záměna = a ==, ...
  - **IndentationError** – špatné odsazení
- Chyby za běhu – odhalí překladač (runtime), lze ošetřit pomocí systému výjimek
  - **NameError** – špatné jméno proměnné (překlep v názvu, chybějící inicializace)
  - **TypeError** – nepovolená operace (sčítání čísla a řetězce, ...)
  - **IndexError** – chyba při indexování řetězce, seznamu, ...
  - **ZeroDivisionError** – dělení nulou
  - **ValueError** – chybná (neočekávaná) hodnota
- Logické chyby – chyba v návrhu algoritmu

Syntaktické chyby

```
>>> 1 = x
  File "<stdin>", line 1
    1 = x
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of
>>> (1]
  File "<stdin>", line 1
    (1]
      ^
SyntaxError: closing parenthesis ']' does not match opening parenthesis '('
>>> if x = True:
  File "<stdin>", line 1
    if x = True:
       ^^^^^^^^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Chyby za běhu

```
>>> a = b + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> 2 + [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Ošetření chyb za běhu – blok `try` – `except`

```
try:
  # blok prikazu
except jmeno_prvni_vyjimky:
  # blok prikazu
except jmeno_dalsi_vyjimky:
  # blok příkazů
# zde je bud konec, nebo zachyceni dalsich vyjimek
```

- Pokud chceme zachytit i chybovou zprávu, musíme napsat:

```
except jmeno_vyjimky as chyba:
  # text vyjimky se ulozi do promenne chyba
```

Více o výjimkách: `https://docs.python.org/3/library/exceptions.html`

Příklad ošetření výjimky

```
>>> (x,y) = (5,0)

>>> z = x/y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> try:
...     z = x/y
... except ZeroDivisionError:
...     print("divide by zero")
...
divide by zero
```

Příklad využití výjimky

```
>>> def nacti_cislo ():
...     spatne = True
...
...     while spatne:
...         try:
...             cislo = float(input("float: "))
...             spatne = False
...         except ValueError as err:
...             print(err)
...         else:
...             return cislo
```

Logické chyby

```
>>> def my_bad_factorial(n):
...     out = 0
...     for i in range(1, n+1):
...         out = out*i
...
...     return out
...
>>> my_bad_factorial(4)
0
>>> my_bad_factorial(10)
0
```

Kde je chyba?

Jak chybám předcházet?

- Pište programy strukturovaně, přemýšlejte i nad možným rozšířením
- Testujte možné varianty průběhu včetně mezních případů
- Pište kód přehledně

Například

```
>>> x = 2
>>> y = x**2 + 2*x+1
```

vypadá o něco lépe než

```
>>> y=x**2
>>> y=y+2*x
>>> y=y+1
```

```
>>> import numpy as np
>>>
>>> s = 0
>>> a = np.random.rand(10)
>>> for i in range(10):
...     s = s + a[i]
...
```

Jak chybám předcházet?

- Pište programy strukturovaně, přemýšlejte i nad možným rozšířením
- Testujte možné varianty průběhu včetně mezních případů
- Pište kód přehledně

Například

```
>>> x = 2
>>> y = x**2 + 2*x+1
```

vypadá o něco lépe než

```
>>> y=x**2
>>> y=y+2*x
>>> y=y+1
```

```
>>> n = 10
>>> s = 0
>>> a = np.random.rand(n)

>>> for i in range(n):
...     s = s + a[i]
```