

```

270     childpos = rightpos
271     # Move the smaller child up.
272     heap[pos] = heap[childpos]
273     pos = childpos
274     childpos = 2*pos + 1
275     # The leaf at pos is empty now. Put newitem there, and bubble it up

```

Algoritmy a programování

Funkce, rekurze (základy)

```

283     # Follow the path to the root, moving parents down until finding a place
284     # newitem fits.
285     while pos > startpos:
286         parentpos = (pos - 1) >> 1
287         parent = heap[parentpos]
288         if parent < newitem:
289             heap[parentpos] = newitem
290             pos = parentpos
291             continue
292         break
293     heap[pos] = newitem
294
295     def _siftup_max(heap, pos):
296         'Maxheap variant of _siftup'
297         endpos = len(heap)
298         startpos = pos
299         newitem = heap[pos]
300         # Bubble up the larger child until hitting a leaf.
301         childpos = 2*pos + 1    # leftmost child position
302         while childpos < endpos:

```

Vojtěch Vonásek

Department of Cybernetics

Faculty of Electrical Engineering

Czech Technical University in Prague

- Výpočet $1 + 2x^2 - 4x^3$ pro $x \in \{0.5, 1, 2\}$

```
1 v1 = 1 + 2*1**2 - 4*1**3
2 print("f(1)=", v1)
3 v2 = 1 + 2*1/2**2 - 4*1/2**3
4 print("f(1/2)=", v2)
5 v3 = 1 + 2*2**2 - 4*2**3
6 print("f(2)=", v3)
```

```
f(1)= -1
f(1/2)= 1.0
f(2)= -15
```

- Co lze programu vytknout?

Motivační příklad

- Výpočet polynomu provedeme ve funkci
- Vstupem funkce je hodnota x
- Výstupem funkce je hodnota $1 + 2x^2 - 4x^3$
 - Klíčové slovo `return`

```

1 def f(x):
2     return 1 + 2*(x**2) - 4*(x**3)
3
4 value = 1
5 print("f(",value, ")=", f(value) )
6 value = 1/2
7 print("f(",value, ")=", f(value) )
8 value = 2
9 print("f(",value, ")=", f(value) )

```

```

f( 1 )= -1
f( 0.5 )= 1.0
f( 2 )= -23

```

- Funkce je způsob strukturování programu
- Funkce řeší konkrétní úkol
- Funkce lze opakovaně volat
- Použití funkcí zjednodušuje vývoj, debugování (ladění), pochopení programu
 - Stačí opravit chybu jednou (ve funkci), ne v každém volání
 - Funkce jsou (typicky) menší a tudíž jednodušší na pochopení
- Při použití funkcí nedochází k duplikaci kódu
- Funkce má jméno, vstup (argument) a poskytuje výstup (návrátová hodnota)

```
1 def functionName(parameters):  
2     code
```

- Pouze příkazy se stejným odsazením jsou součástí funkce

- Funkce vypisuje vstupní argument mezi “!”

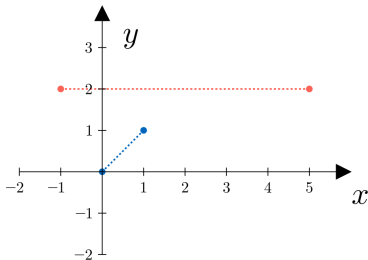
```
1 def Print(text):  
2     print("!!!!!!", text, "!!!!!!")  
3  
4 print("Normal_print")  
5 Print("Access_denied")
```

```
Normal print  
!!!!!! Access denied !!!!!!
```

- Výpočet vzdálenosti 2D bodů (x_1, y_1) a (x_2, y_2)

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
1 def distance(x1,y1,x2,y2):  
2     dx = x1-x2  
3     dy = y1-y2  
4     return (dx**2 + dy**2)**(0.5)  
  
6 print( distance(0,0,1,1) )  
7 print( distance(-1,2,5,2) )
```



```
1.4142135623730951  
6.0
```

- Vstupem funkce je text a písmeno, úkolem je spočítat výskyt tohoto písmena v textu

```
1 def countLetter(text, letter):
2     #text, letter are strings, len(letter) = 1
3     c = 0
4     for i in range(len(text)):
5         if text[i] == letter:
6             c += 1
7     return c
8
9 text = "hippopotomonstrosesquippedaliophobia"
10
11 print( countLetter(text, "o") )
12 print( countLetter(text, "h") )
```

7
2

- Rozšíříme předchozí program na výpočet výskytu všech písmen

```
1 def countLetter(text, letter):
2     #text, letter are strings, len(letter) = 1
3     c = 0
4     for i in range(len(text)):
5         if text[i] == letter:
6             c += 1
7     return c
8
9 text = "hippopotomonstrosesquippedaliophobia"
10 for i in range(ord('a'), ord('z')+1): #note +1 !!!
11     c = countLetter(text, chr(i))
12     if c != 0:
13         print(chr(i), ":", c)
```

a	:	2
b	:	1
d	:	1
e	:	2
h	:	2
i	:	4
l	:	1
m	:	1
n	:	1
o	:	7
p	:	6
q	:	1
r	:	1
s	:	3
t	:	2
u	:	1

Počty parametrů

- žádný parametr
- jeden nebo více parametrů

Návratové hodnoty:

- Jedna nebo více návratových hodnot
- Pokud není explicitně uvedena, je návratová hodnota None

```
1 def f1():
2     a = 1
3                                     #noreturn
4
5 def f2(x):
6     y = x
7                                     #noreturn
8
9 def f3(c,d,a):
10    d = 2
11                                     #noreturn
12
13 def f4():
14     return -1
15
16 def f5(x,y,z):
17     return y,z,y
18
19 f1()
20 f2(10)
21 f3(1,2,3)
22 v = f4()
23 a,b,c = f5(0,1,2)
```

- Funkci musíme volat přesně s tolika argumenty, kolik je vyžadováno v její definici
- (Neplatí v případě argumentů s defaultní hodnotou, ale to nebudeme používat)
- Případné špatné volání je detekováno až za běhu programu!

```
1 def xyz():  
2     print("call_xyz")  
3  
4 xyz()  
5 xyz(12)
```

```
call xyz  
Traceback (most recent call last):  
  File "../functions/function2error.py", line 5, in <  
    module>  
    xyz(12)  
TypeError: xyz() takes 0 positional arguments but 1  
was given
```

- Funkci musíme volat přesně s tolika argumenty, kolik je vyžadováno v její definici
- (Neplatí v případě argumentů s defaultní hodnotou, ale to nebudeme používat)
- Případné špatné volání je detekováno až za běhu programu!

```
1 def awesome(x,y):  
2     return x+y  
3  
4 awesome(12)
```

```
    awesome(12)  
TypeError: awesome() missing 1 required positional  
argument: 'y'
```

- Návratová hodnota je určena klíčovým slovem `return`
- Pokud není explicitně uvedena, je návratová hodnota `None`

```
1 def f1(text):  
2     print("Text: ", text)  
3  
4 a = f1("ahoj")  
5 b = print(a)  
6 print(b)
```

```
Text:  ahoj  
None  
None
```

- Návrátová hodnota je určena klíčovým slovem `return`
- Funkce může vracet více hodnot
- Více návratových hodnot se předává jako typ tuple

```
1 def sortAB(a,b):  
2     if a < b:  
3         return a,b  
4     else:  
5         return b,a  
6  
7 x,y = sortAB(10,-1)  
8 print(x,y)
```

-1 10

- Návrátová hodnota je určena klíčovým slovem `return`
- Funkce může vracet více hodnot
- Více návratových hodnot se předává jako typ tuple

```
1 def sortAB(a,b):  
2     if a < b:  
3         return a,b  
4     else:  
5         return b,a  
6  
7 z = sortAB(10,-1) #all values into tuple z  
8 print(z)  
9 print(type(z))
```

```
(-1, 10)  
<class 'tuple'>
```

- Návrátová hodnota je určena klíčovým slovem `return`
- Volání `return` bez parametru: návratová hodnota je `None`
- Stejně tak pokud je funkce ukončena bez `return`

```
1 def someFunction(x,y):  
2     v = x+y  
3  
4 r = someFunction(10,11)  
5 print(r)
```

`None`

- Datové typy immutable (int, float, string a další) se do funkcí předávají hodnotou
- Změna hodnoty argumentů se neprojeví mimo funkci

```
1 def test1(a,b):  
2     print("Test1", a,b)  
3     a = 0  
4     b = 0  
5     print("Test1", a,b)  
6  
7 x = 10  
8 y = 20  
9 print("Main:",x,y)  
10 test1(x,y)  
11 print("Main:",x,y)
```

```
Main: 10 20  
Test1 10 20  
Test1 0 0  
Main: 10 20
```

- Pokud je proměnná definována ve funkci, je lokální (není přístupná mimo funkci)
- Proměnné definované mimo funkce jsou tzv. globální a funkce je mohou používat **pro čtení**

```
1 def add(x):  
2     x += someValue  
3     return x  
4  
5 someValue = 3  
6 print( add(4) )  
7 print( add(10) )
```

```
7  
13
```

- Pokud je proměnná definována ve funkci, je lokální (není přístupná mimo funkci)
- Proměnné definované mimo funkce jsou tzv. globální a funkce je mohou používat **pro čtení**

```
1 def add(x):  
2     x += someValue  
3     return x  
4  
5 #someValue = 3  
6 print( add(4) )  
7 print( add(10) )
```

```
print( add(4) )  
File "../functions//functionGlobal1.py", line 2, in add  
    x += someValue  
NameError: name 'someValue' is not defined
```

- Pokud je proměnná definována ve funkci, je lokální (není přístupná mimo funkci)
- Proměnné definované mimo funkce jsou tzv. globální a funkce je mohou používat **pro čtení**

```
1 def add(x):  
2     someValue = 1 #new local variable  
3     x += someValue #new local variable  
4     return x  
5  
6 someValue = 3 #global variable  
7 print( add(4) )  
8 print( add(10) )
```

```
5  
11
```

- Proměnná je definována ve funkci $\Rightarrow$ je lokální (není přístupná mimo funkci)
- Proměnné definované mimo funkce jsou tzv. globální a funkce je mohou používat **pro čtení**
- Pokud chceme měnit globální proměnnou $\Rightarrow$ klíčové slovo `global`
- Toto vede na změnu stavu programu — může ovlivnit jiné části programu
- Nepoužíváme

```
1 def add(x):  
2     global someValue  
3     someValue += 1  
4     x += someValue  
5     return x  
6  
7 someValue = 3 #global variable  
8 print("Add", add(4) )  
9 print("someValue", someValue)  
10 print("Add", add(4) )  
11 print("someValue", someValue)
```

```
Add 8  
someValue 4  
Add 9  
someValue 5
```

Pure function (čistá funkce)

- Její výstup závisí (jednoznačně) pouze na vstupních argumentech
- Nemá vedlejší efekty (změna globálních proměnných, souborů)

Pure

```
1 def f1():  
2     return 12  
3  
4 def f2(x,y):  
5     return x*min(x,y)  
6  
7 def f3(x,y,z):  
8     return 123  
9  
10 def f4():  
11     a = 1335
```

Impure

```
1 def f1():  
2     return 12*a #uses global var.  
3  
4 def f2(x,y):  
5     return x*min(x,b) #uses global var.  
6  
7 def f3(x,y,z):  
8     global q #changes global var.  
9     q = x  
10    return x+y+z+q
```

- Omezujeme použití globální proměnných
- Píšeme krátké funkce ($\sim$ jedna obrazovka)
- Jméno funkce vystihuje její obsah
- Jména argumentů takéž
- Preferujeme čisté (pure) funkce

Syntax error

Runtime error

Semantic error

- Při programování se setkáváme s několika typy chyb
- Některé plynou z nesprávného zápisu (syntax error)
- Složitější jsou chyby detekované až za běhu programu (runtime error)
- Sémantické chyby jsou nejtěžší na odhalení

Programming is
10% writing code
and **90%**
understanding why
it's not working.

Chybná syntaxe (syntax error)

- Špatný zápis programu, který porušuje syntaktická pravidla
- Je detekován před spuštěním programu (kontroluje Python)
- Při chybě syntaxe se program nespustí

```
1 def b():  
2     i = 1  
3     print("hello")
```

```
File "../errors//syntaxError1.py", line 3  
    print("hello")  
    ^
```

IndentationError: unexpected indent

```
1 a = 3  
2 b = a**(1/2
```

```
File "../errors//syntaxError2.py", line 3  
    ^
```

SyntaxError: unexpected EOF while parsing

Chybná syntaxe (syntax error)

- Špatný zápis programu, který porušuje syntaktická pravidla
- Je detekován před spuštěním programu (kontroluje Python)
- Při chybě syntaxe se program nespustí

```
1 for i in range(5)
2     print(i)
```

```
File "../errors//syntaxError3.py", line 1
    for i in range(5)
                    ^
```

SyntaxError: invalid syntax

```
1 s = "ahoj'"
2 print(s)
```

```
File "../errors//syntaxError4.py", line 1
    s = "ahoj'"
        ^
```

SyntaxError: EOL while scanning string literal

Chyba běhu (runtime error)

- Chyba, která vznikne až při exekuci programu
- Např. dělení nulou, nevhodné typy argumentů funkcí, volání metod polí na ne-pole proměnné ...
- Tyto chyby nelze detekovat dokud nenastanou

```
1 def gcd(a,b):  
2     return a  
3  
4 print("Program start")  
5 print(gdc(10,2))
```

```
Program start  
Traceback (most recent call last):  
  File "../errors//runtimeError1.py", line 5, in <module>  
    print(gdc(10,2))  
NameError: name 'gdc' is not defined
```

Chyba běhu (runtime error)

- Chyba, která vznikne až při exekuci programu
- Např. dělení nulou, nevhodné typy argumentů funkcí, volání metod polí na ne-pole proměnné ...
- Tyto chyby nelze detekovat dokud nenastanou

```
1 def normalize(a,b,c):  
2     s = a+b+c  
3     return a/s, b/s, c/s  
4  
5 print( normalize(1,2,0) )  
6 print( normalize(1,2,-3) )
```

```
(0.3333333333333333, 0.6666666666666666, 0.0)  
Traceback (most recent call last):  
  File "../errors//runtimeError5.py", line 6, in <module>  
    print( normalize(1,2,-3) )  
  File "../errors//runtimeError5.py", line 3, in normalize  
    return a/s, b/s, c/s  
ZeroDivisionError: division by zero
```

Sémantická chyba (semantic error)

- Program pracuje bez chyby (bez runtime/syntax error), ale jeho výsledek neodpovídá zamýšlenému významu
- Analýza programátorem

```
1 def getMin(a,b):  
2     if a<b:  
3         return a  
4     return b  
5  
6 x = input()  
7 y = input()  
8 print("Mensi_cislo_je", getMin(x,y))
```

inputFromFile.txt

```
12  
5
```

```
python3 semanticError1.py < inputFromFile.txt
```

```
Mensi cislo je 12
```

Rekurze: definice problému pomocí jednodušší varianty stejného problému, odkaz sama na sebe



Implementační pohled (tato přednáška)

- Rekurzivní volání je pokud funkce volá sebe samu
- Více funkcí se volá navzájem

Algoritmický pohled (další přednášky)

- Rekurze je způsob řešení problémů
- Rozděl a panuj (Divide and Conquer)
 - řešení problému je založeno na řešení jednodušší varianty stejného problému
- Rekurzivní definice matematických funkcí

```

1 def f(x):
2     return f(x-1)
3
4 def a(x):
5     b(x)
6
7 def b(x):
8     a(x)
    
```

$$x_{n+1} = rx_n(1 - x_n)$$

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = F_1 = 1$$

- Funkce volá samu sebe
- Bez ukončovací podmínky vzniká nekonečný cyklus
- Prakticky je počet volání omezen operačním systémem ($\sim 10^3$ volání)
- Překročení tohoto limitu vede na chybu programu

```
1 def f(x):      #nekonecna rekurze
2     return f(x-1)
3
4 f(10)
```

```
f(x)
[Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

- Při použití rekurze je třeba vždy uvést ukončovací podmínku

- Při použití rekurze je třeba vždy uvést ukončovací podmínku

```
1 def f(n):  
2     if n > 0:  
3         return 1+f(n-1)  
4     return 0  
5  
6 print( f(-1) )  
7 print( f(0) )  
8 print( f(5) )  
9 print( f(6) )
```

```
0  
0  
5  
6
```

- Argument se snižuje, $f(n) \rightarrow f(n-1) \rightarrow f(n-2) \rightarrow \dots$
- Je třeba zespoda omezit na N_{min}

```
1 def f(n):  
2     if n > Nmin:  
3         f(n-1)  
4     else  
5         return
```

- Argument se zvyšuje, $f(n) \rightarrow f(n+1) \rightarrow f(n+2) \rightarrow \dots$
- Je třeba shora omezit na N_{max}

```
1 def f(x):    #pocet volani je omezen  
2     if x < Nmax:  
3         return f(x+1)  
4     else:  
5         return
```

- Přímá definice: $x^n = \prod_{i=1}^n x = \underbrace{x \cdot x \cdots x}_n$
- Rekurzivní definici: $x^n = x \cdot x^{n-1}$
- Základní případ: $x^0 = 1$

```
1 def prod_iterative(x,n):  
2     result = 1  
3     for _ in range(n):  
4         result = result * x  
5     return result  
6  
7  
8 print(prod_iterative(10,0))  
9 print(prod_iterative(2,10))  
10 print(prod_iterative(2**(0.5),2))
```

```
1  
1024  
2.000000000000000004
```

- Přímá definice: $x^n = \prod_{i=1}^n x = \underbrace{x \cdot x \cdots x}_n$
- Rekurzivní definici: $x^n = x \cdot x^{n-1}$
- Základní případ: $x^0 = 1$

```
1 def prod_recursive(x,n):  
2     if n == 0:  
3         return 1  
4     return x * prod_recursive(x,n-1)  
5  
6 print(prod_recursive(10,0))  
7 print(prod_recursive(2,10))  
8 print(prod_recursive(2**(0.5),2))
```

```
1  
1024  
2.000000000000000004
```

- Vstup je řetězec, úkolem je uložit (vypsát) ho v opačném pořadí
- ahoj → joha

Klasické řešení přes cyklus

- Projdeme string pozpátku, např. for cyklem

```
1 def reverseIterative(x):  
2     result = ""  
3     for i in range(len(x)-1,-1,-1):  
4         result += x[i]  
5     return result  
6  
7 print( reverseIterative("PYTHON") )
```

NOHTYP

- Vstup je řetězec, úkolem je uložit (vypsát) ho v opačném pořadí
- ahoj → joha

Rekurzivní přístup

- Základní případ: pokud vstup je prázdný string, vrátíme prázdný string
- Jinak: otočíme znaky `x[1:]` a přidáme k nim první znak `x[0]`

```
1 def reverseRecursive(x):  
2     if len(x) == 0:  
3         return ""  
4     return reverseRecursive(x[1:]) + x[0]  
5  
6 print( reverseRecursive("PYTHON") )
```

NOHTYP

- Existuje řetězec, kde rekurzivní řešení selže?

- Každý cyklus lze nahradit rekurzí

Klasický cyklus $0, 1, \dots, n-1$

```
1 def countUp(n):  
2     for i in range(n):  
3         print(i)  
4  
5 countUp(5)
```

```
0  
1  
2  
3  
4
```

- Každý cyklus lze nahradit rekurzí

Rekurzivní přístup výpočtu $0, 1, \dots, n-1$

- Vnitřní funkce `countUpInner` volá sebe sama dokud aktuální hodnota je menší než maximum
- Uživatel může zavolat buď `countUpInner(n,0)` nebo `countUpRecursive(n)`

```
1 def countUpInner(maxvalue, actualvalue):  
2     if actualvalue < maxvalue:  
3         print(actualvalue)  
4         countUpInner(maxvalue, actualvalue+1)  
5  
6 def countUpRecursive(n):  
7     countUpInner(n,0)  
8  
9 countUpRecursive(5)
```

0
1
2
3
4

- Jaká je nevýhoda rekurzivního řešení oproti `for+range` ?

- Přímá definice: $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- Rekurzivní definice: $n! = n \cdot (n-1)!$
- Součástí rekurzivní definice je tzv. základní (bázový) případ: $0! = 1! = 1$
- Základní případ slouží jako ukončovací podmínka

```
1 def fact(n):  
2     if n == 0 or n == 1: #basic case  
3         return 1  
4     return n * fact(n-1)  
5  
6 for i in range(6):  
7     print(i, "!=", fact(i), sep=" ")
```

```
0! = 1  
1! = 1  
2! = 2  
3! = 6  
4! = 24  
5! = 120
```

- Srinivasa Ramanujan odvodil

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)! (1103 + 26390k)}{(k!)^4 396^{4k}}$$

- Napište funkci, která realizuje tento výpočet
- Jaké podpůrné operace budou k výpočtu potřeba?
- Jaké typy proměnných budeme potřebovat (float, string, int?)

Příklady funkcí: aproximace π

- Opakující se operace (faktoriál) realizujeme funkcí, sumu jako for cyklus

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{\overbrace{(4k)! (1103 + 26390k)}^a}{\underbrace{(k!)^4 396^{4k}}_b}$$

```

1 def fact(n):
2     res = 1
3     for i in range(2, n+1):
4         res *= i
5     return res
6
7 def piApprox():
8     s = 0           #for computing the sum
9     kmax = 6        #what is good value?
10    for k in range(kmax):
11        a = fact(4*k)* (1103 + 26390*k)
12        b = (fact(k)**4)*(396**(4*k))
13        s += a/b
14        print(k, a/b)    #debug output
15    r = (2*(2**(0.5)) / 9801)*s
16    return 1/r
17
18 print( "PI=", piApprox() )

```

Výpis volání piApprox:

```
0 1103.0
1 2.6831974348925308e-05
2 2.2453850201136644e-13
3 1.995074994495897e-21
4 1.8393545314677564e-29
5 1.7358835411230251e-37
PI= 3.141592653589793
```

- Výsledek a/b rychle klesá, ve 4. iteraci je $\sim 1.8 \cdot 10^{-29}$
- Prakticky nemá smysl dělat více než několik iterací
- Otestujte program pro různá $kmax$. Co se stane, pokud $kmax = 1$?
- Existuje $kmax$, pro které program selže?