Algoritmy a programování

Abstraktní datové struktury: listy

Vojtěch Vonásek

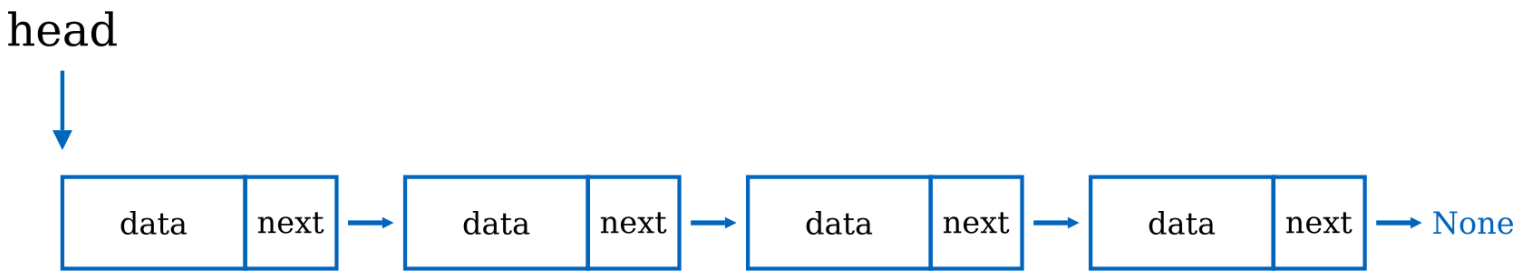
Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

Linked List — (lineární) spojový seznam

- Složená datová struktura
- Obsahuje předem neznámý počet položek

Princip

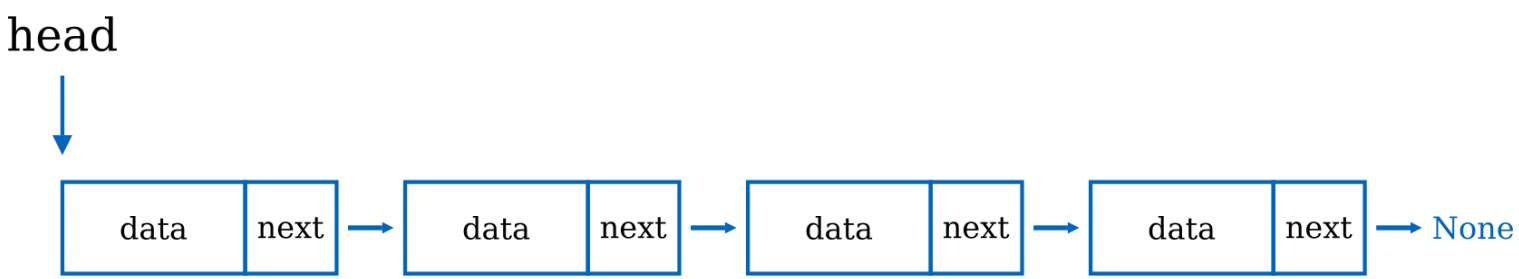
- Data jsou uložena v uzlech (co položka, to uzel)
- Uzly ukazují na následníka (poslední uzel neukazuje nikam — `None`)
- Spojový seznam je reprezentován ukazatelem na první prvek (head, first, begin, . . .)



Linked List: implementace

- Uzel je reprentován třídou `Node`
- Spojový list je třída `LinkedList`, která obsahuje položku `head`

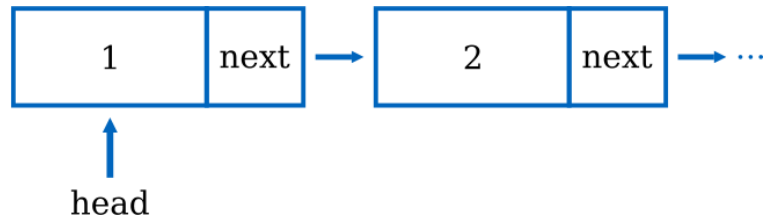
```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class LinkedList:
7     def __init__(self):
8         self.head = None
```

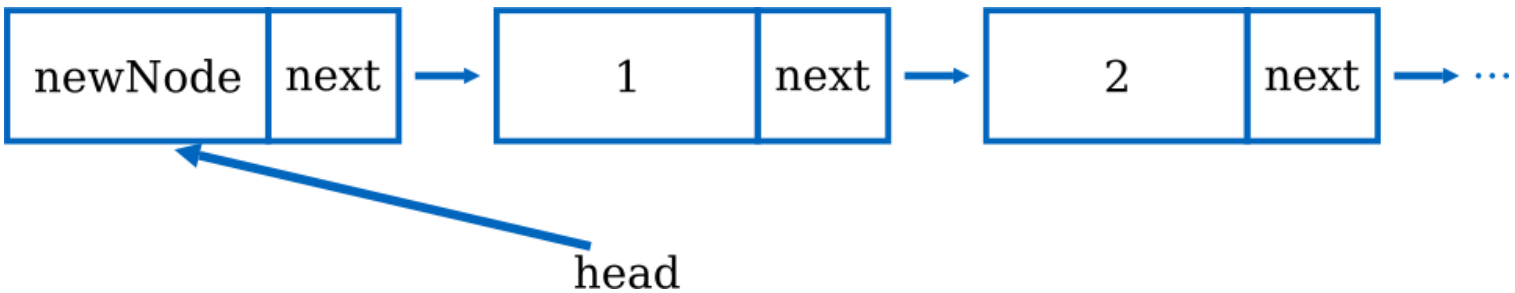


Přidání na začátek seznamu

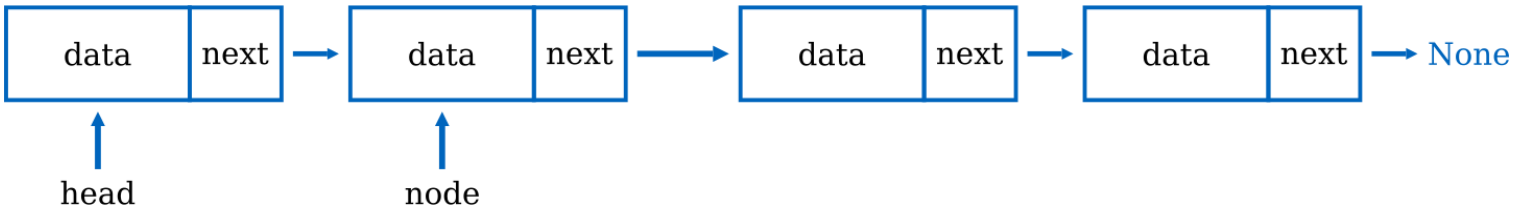
- Vytvoříme nový prvek, jeho next ukazuje na head
- Head nastavíme na nový prvek

```
1     def add(self, data):
2         newNode = Node(data)
3         newNode.next = self.head
4         self.head = newNode
```



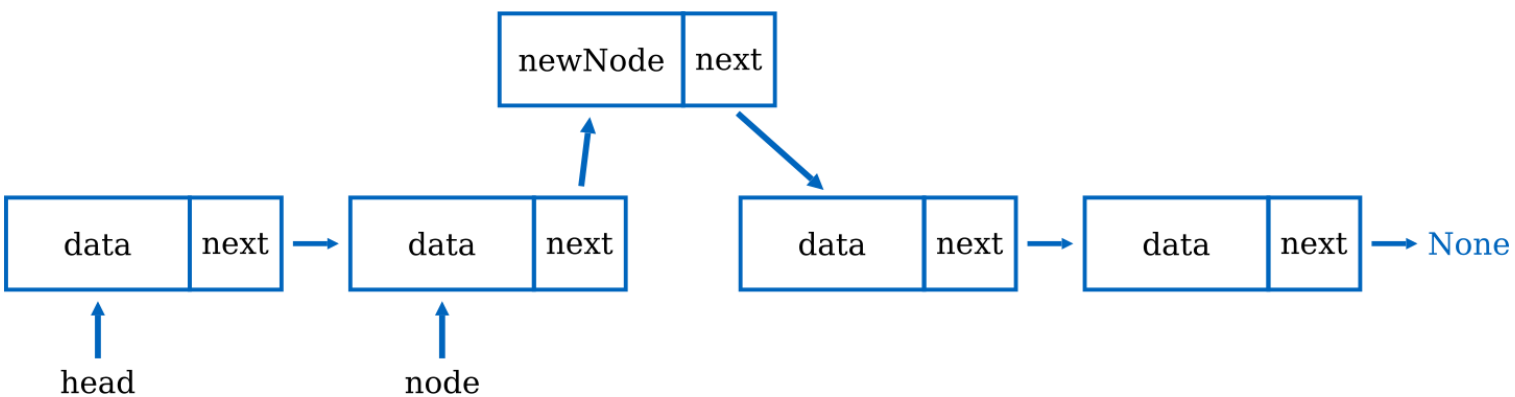


Přidání za libovolný prvek `node`



- Vytvoříme nový prvek, jeho next ukazuje na `node.next`
- `node.next` ukazuje na nový prvek

```
1     def insert(self, data, node):
2         newNode = Node(data)
3         newNode.next = node.next
4         node.next = newNode
5         return newNode
```



Odebrání prvku

- Implementace funkce `pop()`: odebrat prvek, vrátit jeho data
- Uložíme data prvku `head`
- `head` přesměrujeme na další prvek: `head = head.next`
- Vrátíme data

```
1     def pop(self):
2         #assuming head is not None
3         data = self.head.data
4         self.head = self.head.next
5         return data
```

Procházení seznamu

- Procházíme od `head` tak, že následujeme prvky node.next

```
1     def print(self):
2         tmp = self.head
3         while tmp != None:
4             print(tmp.data)
5             tmp = tmp.next
```

Linked List: příklad použití


```
1  from linkedList import LinkedList
2
3  llist = LinkedList()
4  for i in range(5):
5      llist.add(i)
6
7  llist.print()
8  quit()
9
10 data = llist.pop()
11 print("Pop", data)
12 data = llist.pop()
13 print("Pop", data)
14 llist.print()
```

```
4
3
2
1
0
```

Linked List: příklad použití


```
1  from linkedList import LinkedList
2
3  llist = LinkedList()
4  for i in range(5):
5      llist.add(i)
6
7  llist.print()
8  quit()
9
10 data = llist.pop()
11 print("Pop", data)
12 data = llist.pop()
13 print("Pop", data)
14 llist.print()
```

```
4
3
2
1
0
```

Linked List: hledání prvku

linkedList.py

```
1     def find(self, data):
2         tmp = self.head
3         while tmp != None:
4             if tmp.data == data:
5                 return tmp
6             tmp = tmp.next
7         return None
```

```
1  from linkedList import LinkedList
2
3  llist = LinkedList()
4  for i in range(5):
5      llist.add(i)
6  llist.print()
7
8  node = llist.find(2)
9  print(node)
10 print(node.data)
```

```
4
3
2
1
0
<linkedList.Node object at
    0x7fb8f3b0b940>
2
```

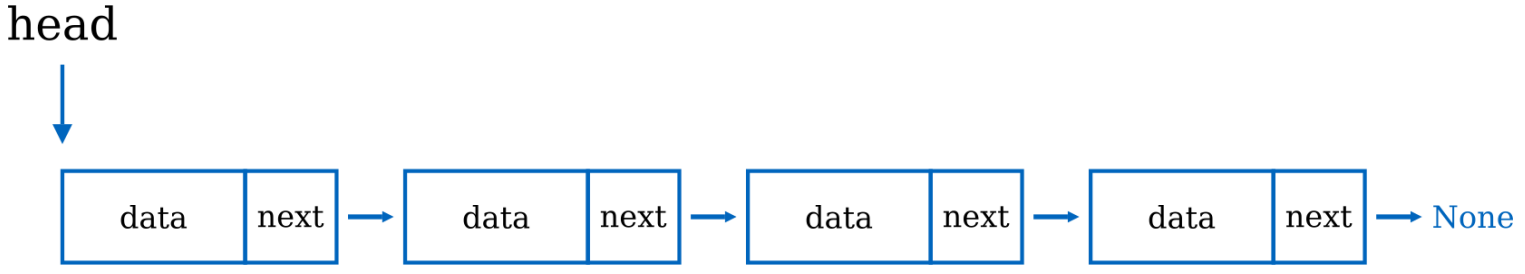
Linked List

Vlastnosti

- Spojový seznam umožňuje rychle přidávat/mazat uzly
- Nepodporuje přistup na libovolnou položku (random access)
- Položky lze procházet jen v jednom směru

Častová složitost

- Přidání/vložení/smazání uzlu: $\mathcal{O}(1)$
- Hledání prvku: $\mathcal{O}(n)$
- Přístup na i-tý prvek: $\mathcal{O}(i) = \mathcal{O}(n)$



Použití

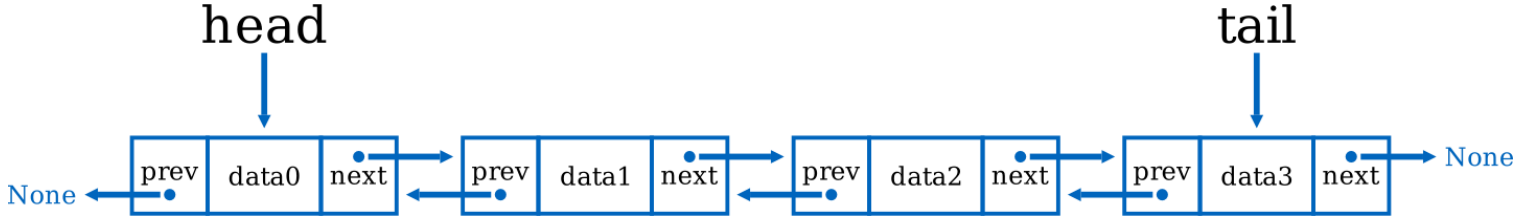
- Uchování dat jejichž počet dopředu neznáme (a není třeba přímý přístup)
- např. pro realizaci zásobníku, grafové algoritmy

Double Linked List — obousměrný spojový seznam

- Složená datová struktura
- Obsahuje předem neznámý počet položek

Princip

- Data jsou uložena v uzlech (co položka, to uzel)
- Uzly ukazují na následníka a na předchůdce
- Double Linked list je reprezentován ukazatelem na začátek (head) a na poslední prvek (tail)

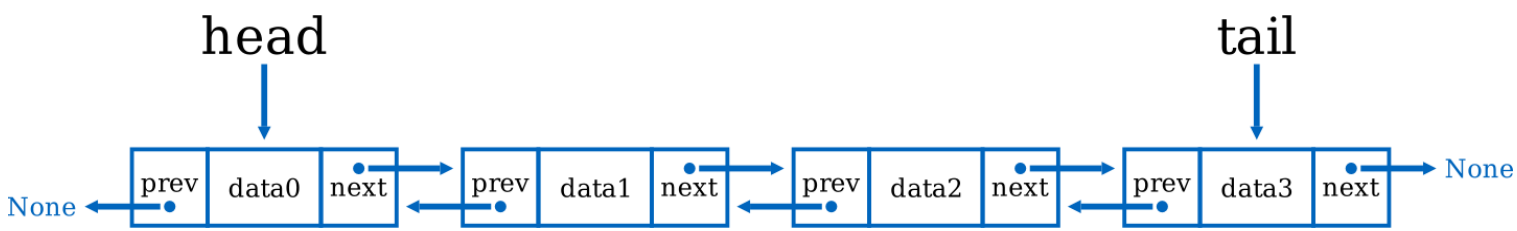


Double Linked List: implementace

- Uchováváme referenci na první (head) a poslední prvek (tail)
- Pokud prvek neexistuje, používáme `None`

`doubleLinkedList.py`

```
1  class Node:
2      def __init__(self, data):
3          self.prev = None
4          self.next = None
5          self.data = data
6
7  class DoubleLinkedList:
8      def __init__(self):
9          self.head = None
10         self.tail = None
```



Procházení seznamu

- Dopředný průchod: (forward), procházíme list od začátku po následnících
- Zpětný průchod: (backward), procházíme list od kone po předchůdcích

doubleLinkedList.py

```
1     def traverseForward(self, fromNode = None):
2         if fromNode == None:
3             fromNode = self.head
4         while fromNode != None:
5             print(fromNode.data)
6             fromNode = fromNode.next
7
8     def traverseBackward(self, fromNode = None):
9         if fromNode == None:
10            fromNode = self.tail
11        while fromNode != None:
12            print(fromNode.data)
13            fromNode = fromNode.prev
```

Přídání za existující uzel

- Vytvoříme nový uzel `newNode` a chceme jej vložit za existující uzel `node`
- Pokud je `node` na konci seznamu, stane se `newNode` novým koncem (tail)
- Jinak upravíme `newNode`, aby ukazoval na následníka `node`, a tohoto následníka tak, aby ukazoval na `newNode`
- Následníkem uzlu `node` se stane uzel `newNode`

`doubleLinkedList.py`

```
1     def addAfter(self, node, newData):
2         newNode = Node(newData)
3         newNode.prev = node
4         if node.next == None:
5             newNode.next = None
6             self.tail = newNode
7         else:
8             newNode.next = node.next
9             node.next.prev = newNode
10        node.next = newNode
```

Přídání před existující uzel

- Vytvoříme nový uzel `newNode` a chceme jej vložit před existující uzel `node`
- Pokud je `node` na začátku seznamu, stane se `newNode` novým začátkem (head)
- Jinak upravíme `newNode`, aby ukazoval na předchůdce `node`, a tohoto předchůdce tak, aby ukazoval na `newNode`
- Předchůdcem uzlu `node` se stane uzel `newNode`

`doubleLinkedList.py`

```
1     def addBefore(self, node, newData):
2         newNode = Node(newData)
3         newNode.next = node
4         if node.prev == None:
5             newNode.prev = None
6             self.head = newNode
7         else:
8             newNode.prev = node.prev
9             node.prev.next = newNode
10        node.prev = newNode
```

Přidání prvku na začátek prázdného seznamu

- Předchozí operace `addAfter` a `addBefore` předpokládají neprázdný seznam
- Pro vložení prvku do prázdného seznamu musíme ještě inicializovat `head` a `tail`

`doubleLinkedList.py`

```
1     def addBegin(self, newData):
2         if self.head == None:
3             newNode = Node(newData)
4             self.head = newNode
5             self.tail = newNode
6             newNode.next = None
7             newNode.prev = None
8         else:
9            self.addBefore(self.head, newData)
```

Přidání prvku na konec prázdného seznamu

- Předchozí operace `addAfter` a `addBefore` předpokládají neprázdný seznam
- Pro vložení prvku do prázdného seznamu musíme ještě inicializovat `head` a `tail`

`doubleLinkedList.py`

```
1     def addEnd(self, newData):
2         if self.tail == None:
3             self.addBegin(newData)
4         else:
5             self.addAfter(self.tail, newData)
```

Double Linked List: příklad práce s listem


```
1  from doubleLinkedList import *
2
3  mylist = DoubleLinkedList()
4
5  mylist.addBegin("0")
6  mylist.addBegin("1")
7  mylist.addEnd("end")
8  mylist.addEnd("end2")
9
10 mylist.traverseForward()
11 print()
12 mylist.traverseBackward()
```

```
1
0
end
end2

end2
end
0
1
```

Double Linked List: příklad vyhledání prvku

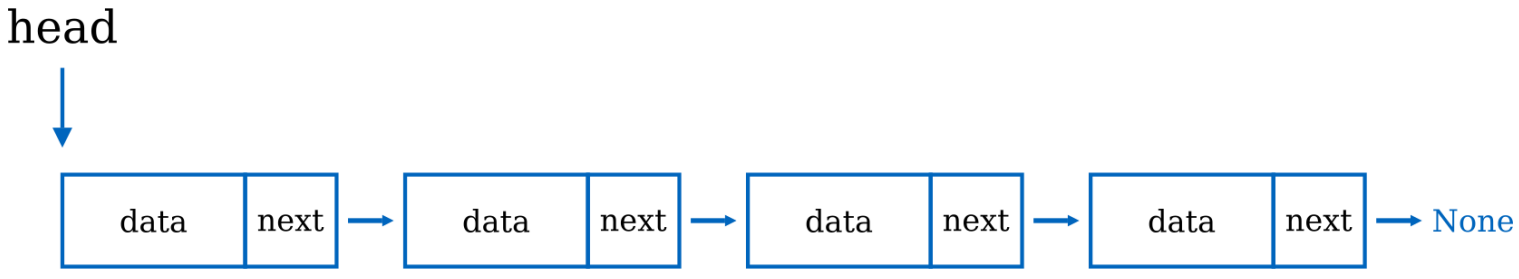

```
1  from doubleLinkedList import *
2
3  mylist = DoubleLinkedList()
4  for i in range(5):
5      mylist.addBegin(i)
6
7  mylist.traverseForward()
8
9  numberToFind = 3
10 tmp = mylist.head
11 while tmp != None:
12     if tmp.data == numberToFind:
13         break
14     tmp = tmp.next
15
16 if tmp != None:
17     print("Found item", tmp.data)
18     print("tmp=", tmp)
19     mylist.addAfter(tmp, "3.5")
20 mylist.traverseForward()
```

```
4
3
2
1
0
Found item 3
tmp= <doubleLinkedList.
    Node object at 0
    x7f91433afbb0>
4
3
3.5
2
1
0
```

Double Linked List

Častová složitost

- Přidání/vložení/smazání uzlu na začátku/konci/(v uzlu*): $\mathcal{O}(1)$
- Hledání prvku: $\mathcal{O}(n)$
- Přístup na i-tý prvek: $\mathcal{O}(i) = \mathcal{O}(n)$
- * pokud uzel už známe



Použití

- např. pro realizaci zásobníku a fronty
- uložení stavů (např. www browser, textový editor)

List vs. pole v Pythonu

Pole

- Pevně daná délka, je dopředu známa, velikost pole nejde měnit
- Rychlý $\mathcal{O}(1)$ přístup na libovolný prvek — operátor `[]`
- Součástí většiny jazyků, c,c++, ...

Dynamické pole

- Velikost pole není dopředu známa, pole se (re)alokuje dynamicky při přidání/odebrání prvků (*)
- Rychlý $\mathcal{O}(1)$ přístup na libovolný prvek — operátor `[]`
- C nemá, C++ jako součást standard template library
- V Python se tomuto datovému typu říká List
- * Typicky se vnitřně se alokuje 2x více, než aktuální velikost

List (linked nebo double linked)

- Počet prvků není dopředu znám, rychlé mazání a přidání
- Pomalý přístup na i-tý prvek
- V Pythonu jako součást různých knihoven

Moduly v Pythonu

- Samostatný soubor s proměnnými a funkcemi
- Knihovny jsou kolekce modulů
- Přístup na data (funkce, proměnné) modulů přes `import`

`mymodule.py`

```
1  a = 123
2
3  def fact(n):
4      s = 1
5      for i in range(2,n):
6          s *= i
7      return s
8
9  def avg(data):
10     return sum(data)/len(data)
```

`test1.py`

```
1  import mymodule
2
3  for i in range(5):
4      print(fact(i))
```

```
Traceback (most recent call last):
  File "../modules//test1.py", line 4, in <module>
    print(fact(i))
NameError: name 'fact' is not defined
```

Moduly v Pythonu

- Samostatný soubor s proměnnými a funkcemi
- Knihovny jsou kolekce modulů
- Přístup na data (funkce, proměnné) modulů přes `import`

`mymodule.py`

```
1  a = 123
2
3  def fact(n):
4      s = 1
5      for i in range(2,n):
6          s *= i
7      return s
8
9  def avg(data):
10     return sum(data)/len(data)
```

`test1.py`

```
1  import mymodule
2
3  for i in range(5):
4      print(mymodule.fact(i))
5
6  print("Module a=", mymodule.a)
```

```
1
1
1
2
6
Module a= 123
```

Moduly v Pythonu

- Samostatný soubor s proměnnými a funkcemi
- Knihovny jsou kolekce modulů
- Přístup na data (funkce, proměnné) modulů přes `import`

`mymodule.py`

```
1  a = 123
2
3  def fact(n):
4      s = 1
5      for i in range(2,n):
6          s *= i
7      return s
8
9  def avg(data):
10     return sum(data)/len(data)
```

`test1.py`

```
1  from mymodule import *
2
3  for i in range(5):
4      print(fact(i))
5
6  print("Module a=", a)
```

```
1
1
1
2
6
Module a= 123
```

Moduly v Pythonu

- Samostatný soubor s proměnnými a funkcemi
- Knihovny jsou kolekce modulů
- Přístup na data (funkce, proměnné) modulů přes `import`

`mymodule.py`

```
1  a = 123
2
3  def fact(n):
4      s = 1
5      for i in range(2,n):
6          s *= i
7      return s
8
9  def avg(data):
10     return sum(data)/len(data)
```

`test1.py`

```
1  import mymodule as M
2
3  for i in range(5):
4      print(M.fact(i))
5
6  print("Module a=", M.a)
```

```
1
1
1
2
6
Module a= 123
```

Moduly v Pythonu

- Moduly v adreářích importujeme včetně jejich cesty (používáma '.' pro oddělení adresářů)

mylibrary/functions.py

```
1
2 def f1():
3     print("F1")
4
5 def f2():
6     print("F2")
```

test1.py

```
1 import mylibrary.functions as F
2
3 F.f1()
```

```
F1
```