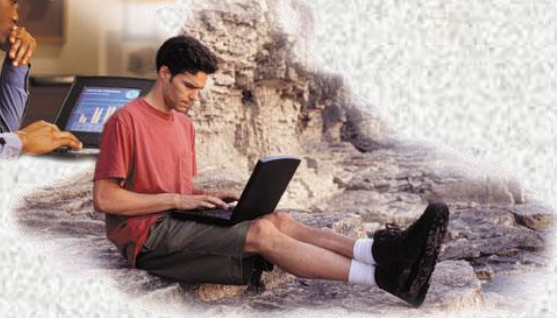




# Основи на Програмирането



## Лекция 4



Изрази и присвояване. Условни команди и оператори. Списъци и редици.

# Какво ще научите

- Изразите като команди
- Приоритет на операторите
- Многозначно и кратко присвояване
- Разклоняване на изпълнението с условна команда
- Условен оператор за изпълнение
- Списъци
- Оператори и функции за работа със списъци
- Редици
- Оператори и функции за работа с редици

# Програми на PYTHON

- **Всяка програма съдържа модули**
- **Всеки модул съдържа команди**
- **Командите съдържат изрази**
- **Изразите създават и обработват обекти**

## Изрази в PYTHON

- **Изразът** представлява обекти свързани с оператори
- В най-простия си вид изразът е обект или име
- **Изразът** е най-простият вид команда – допуска се задаване на израз на отделен ред
- **Изразът** се оценява до някаква конкретна стойност от някакъв тип
- В хода на оценяването е възможно настъпване на страничен ефект (промяна на стойност на обект от израза)



## Изрази в PYTHON - 2

- **Изразът** който е пресметнат последно в интерпретатора има стойност, която се съхранява в променлива с име “\_”
- **Изразите** се използват като стойност в оператор, за обръщение към функция, за извеждане на стойност на екрана
- **Изразът** `print(<аргументи>)` извежда аргументите на екрана или във файл и се оценява до стойност `None`

## Анонимна променлива

```
>>> 5+7*4
```

```
33
```

```
>>> _
```

```
33
```

```
>>> x = _ * 2
```

```
>>> x
```

```
66
```

```
>>>
```

## Израз за извеждане на стойности

```
>>> x = print('Spam')
```

Spam

*# print е вградена функция за извеждане*

```
>>> print(x)  # Но като израз връща стойност
```

None

## Изрази със страничен ефект

```
>>> L = [1, 2]
```

```
>>> L.append(3)    # методът append променя L
```

```
>>> L
```

```
[1, 2, 3]
```

```
>>> Res = L.append(4) # но append връща None, не L
```

```
>>> print(Res)
```

```
None
```



# Оператори

В езика Python има няколко групи оператори:

- ✓ Аритметични
- ✓ За сравнение
- ✓ Логически
- ✓ Побитови
- ✓ За присвояване
- ✓ Специални

# Аритметични оператори

- + -> оператор за събиране
- -> оператор за изваждане
- \* -> оператор за умножение
- % -> оператор за остатък при целочислено деление
- // -> оператор за целочислено деление
- / -> оператор за деление
- \*\* -> оператор за повдигане в степен

# Оператори за сравнение

> -> оператор за по-голямо

< -> оператор за по-малко

== -> оператор за равна стойност

!= -> оператор за различна стойност

>= -> оператор за по-голямо или равно

<= -> оператор за по-малко или равно

## Логически оператори

`and` -> връща втория аргумент ако първият е истина, иначе връща първия аргумент

`or` -> връща първия аргумент ако той е истина, иначе връща втория аргумент

`not` -> има един аргумент и връща истина, ако аргумента е лъжа, или лъжа ако аргумента е истина

# Логически оператори - пример

```
>>> 5 and 7
```

```
7
```

```
>>> 5 or 7
```

```
5
```

```
>>> not 5
```

```
False
```

```
>>> not 0
```

```
True
```

```
>>> '123' and '456'
```

```
'456'
```



## Побитови оператори

& -> логическо И на всяка двойка битове

| -> логическо ИЛИ на всяка двойка битове

~ -> промяна на всеки бит в обратния

^ -> изключващо ИЛИ на всяка двойка битове

>> -> изместване на битовете в дясно

<< -> изместване на битовете в ляво

# Побитови оператори - пример

```
>>> 5 & 7
```

```
5
```

```
>>> 5 | 7
```

```
7
```

```
>>> 5 ^ 7
```

```
2
```

```
>>> ~7
```

```
-8
```

```
>>> 7 >> 2
```

```
1
```

```
>>> 7 << 2
```

```
28
```

## Побитови оператори – пример 2

```
>>> 5 and 7
```

```
7
```

```
>>> 7 and 5
```

```
5
```

```
>>> 7 & 5
```

```
5
```

```
>>> 5 & 7
```

```
5
```

## Специални оператори

- ✓ `is` -> истина ако и двата аргумента сочат един и същи обект (ако са еднакви)
- ✓ `is not` -> истина ако и двата аргумента не сочат един и същи обект (ако са различни)
- ✓ `in` -> истина ако първия аргумент се съдържа в контейнера втори аргумент
- ✓ `not in` -> истина ако първия аргумент не се съдържа в контейнера втори аргумент

# Приоритет на операторите в изразите

Операторите подредени в низходящ ред по приоритет:

1.  $-x$ ,  $+x$ ,  $\sim x$ ,  $**$
2.  $*$ ,  $\%$ ,  $/$ ,  $//$
3.  $+$ ,  $-$
4.  $<<$ ,  $>>$
5.  $\&$
6.  $^$
7.  $|$
8.  $==$ ,  $!=$ ,  $>$ ,  $>=$ ,  $<$ ,  $<=$ ,  $is$ ,  $is\ not$ ,  $in$ ,  $not\ in$
9.  $not$
10.  $and$
11.  $or$
12.  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $//=$ ,  $\%=$ ,  $**=$



# Приоритет на операторите в изразите

- ✓ В един израз поредността на прилагане на операторите се определя от наличието на скоби
- ✓ При липса на скоби, операторите се прилагат в ред от най-високия приоритет към най-малкия:
- ✓  $3+4*5 = 3 + (4 * 5) = 23$
- ✓ При последователност от оператори с еднакъв приоритет, те се изпълняват последователно от ляво на дясно

# Команда за присвояване

Синтаксис:

NAME = <израз>

Семантика:

- (1) Оценява зададения израз <израз> до обект с някаква стойност
- (2) Свързва променливата NAME с обекта с получената стойност

$x = 23 + 5$       # променливата x свързваме с обекта  
# цяло число 28

# Оператори за съкратено присвояване

$x \text{ <op>= } y \rightarrow x = x \text{ <op> } y$

$x += 7 \rightarrow x = x + 7$

$x \&= 5 \rightarrow x = x \& 5$

Команди за присвояване:

➤ `size = 40`

➤ `a = b = c = 3`

➤ `var1, var2 = var2, var1`

# Примери за кратко присвояване

`Suma += 21` # екв. `Suma = Suma + 21`

Прилага се при следните оператори:

`X += Y`   `X &= Y`   `X -= Y`   `X |= Y`

`X *= Y`   `X ^= Y`   `X /= Y`   `X >>= Y`

`X %= Y`   `X <<= Y`   `X **= Y`   `X //= Y`

# Смисъл на кратко присвояване

Изпълнява се по-бързо и ефективно от нормалното (обектът в ляво се оценява само един път)

Прилага се по различен начин в зависимост от типа на обектите

Резултатът е различен за mutable / immutable обекти



# Оператор-израз за присвояване

Синтаксис:

NAME := <израз>

Семантика:

- (1) Оценява зададения израз <израз> до някаква стойност
- (2) Присвоява получената стойност на променливата NAME
- (3) Тази стойност е и стойността на оператора-израз за присвояване

# Оператор-израз за присвояване 1

```
my_list = [1,2,3]
```

```
count = len(my_list)
```

```
if count > 3:
```

```
    print(f " Грешка, {count} са твърде много")
```

Чрез оператор-израз за присвояване:

```
my_list = [1,2,3]
```

```
if (count := len(my_list)) > 3:
```

```
    print(f" Грешка, {count} са твърде много ")
```

## Оператор-израз за присвояване 2

```
n = 1
```

```
while n < 3:
```

```
    print(n) # 1,2
```

```
    n += 1
```

Чрез оператор-израз за присвояване:

```
w = 0
```

```
while (w := w + 1) < 3:
```

```
    print(w) # 1,2
```

## Оператор-израз за присвояване 3

while True:

    p = input(" Въведете парола: ")

    if p == "Паролата":

        break

Чрез оператор-израз за присвояване:

while (p := input(" Въведете парола: ")) == "Паролата":

    break

## Оператор-израз за присвояване 4

```
records = api.readFailedRecords()  
if len(records) > 0:  
    for record in records:  
        api.assignToTechnician(record)
```

Чрез оператор-израз за присвояване:

```
if records := api.readFailedRecords():  
    for record in records:  
        api.assignToTechnician(record)
```



# Многозначно присвояване

```
>>> a, b = 5, 6
```

```
>>> a
```

```
5
```

```
>>> b
```

```
6
```

```
>>>
```

# Условна команда if

Общ синтаксис на командата:

if <израз 1>:

    <блок команди 1>

elif <израз 2>: # 0 или повече elif клаузи

    <блок команди 2>

else: # Допуска се 0 или 1 клауза else

    <блок команди 3>

<следващи команди ...>

## Пример за проверки с if

```
x = int(input("Въведете цяло число:"))
if x < 0:
    x = 0
    print('Отрицателно става нула')
elif x == 0:
    print('Нула')
elif x == 1:
    print('Единица')
else:
    print('По-голямо')
```

## Пример за проверки с if

След двоеточието вместо блок команди на следващи редове може да се зададе единична проста команда на същия ред след двоеточието (не се препоръчва):

```
if x < 0:    print('Отрицателно')
elif x == 0: print('Нула')
elif x == 1: print('Единица')
else: print('Повече')
```

# Съкратена форма на if като оператор

```
A = Y if X else Z
```

Тази форма е еквивалентна на:

```
if X:  
    A = Y  
else:  
    A = Z
```



# Съкратена форма на if като оператор

$A = Y \text{ if } X \text{ else } Z$

Тази форма е еквивалентна също и на:

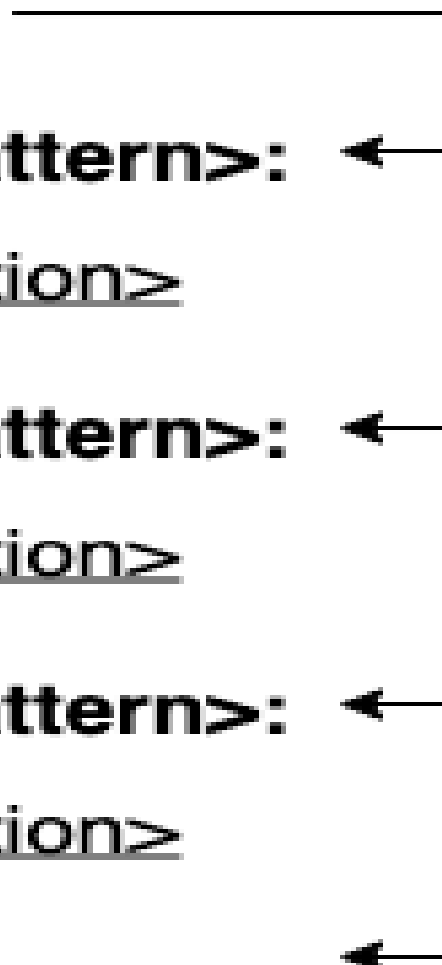
$A = ((X \text{ and } Y) \text{ or } Z)$

Проблем: не може да се използва ако Y има логическа стойност False.

Тогава, ако X има стойност True, според съкратената форма резултатът е False, а според логическия израз е Z

# Структурни проверки с match

**match subject:**



```
case <pattern>: <action>  
case <pattern>: <action>  
case <pattern>: <action>  
case _: <action wildcard>
```

# Структурни проверки с match

```
# status – съдържа код на завършване  
# на изпълнението на http заявка  
match status:  
    case 400:  
        return "Лоша заявка"  
    case 404:  
        return "Не открит ресурс"  
    case 418:  
        return "Програмно зависим"  
    case _:  
        return "Проблем в Интернет"
```

# Структурни проверки с match

# point е редица от 2 елемента (x, y)

match point:

case (0, 0):

print("Начало")

case (0, y):

print(f"Y={y}")

case (x, 0):

print(f"X={x}")

case (x, y):

print(f"X={x}, Y={y}")

case \_:

raise ValueError("Не е точка")

# Съставни блокови команди

- ✓ Започват с основна клауза, завършваща с “:”
- ✓ След нея започва блок от команди, започващи на нов ред и изместени с няколко интервала в дясно в сравнение с основната клауза
- ✓ Блокът завършва с команда, след която има команда на нов ред с по-малко изместване
- ✓ Допуска се команда от блока да бъде съставна блокова команда, което води до още по-големи отмествания на следващите блокове
- ✓ Примери – if ; for ; while ...



# Съставни блокови команди - пример

```
if x == 5:
```

```
    y = x - 1
```

```
    print(y, x)
```

```
elif x > 5:
```

```
    if y > 0:
```

```
        y -= 1
```

```
        x = x - y
```

```
    else:
```

```
        x = x - 1
```

```
else print('End')
```

# Контейнери

- Това са **обекти** които имат като елементи указатели (идентичност) на други обекти

**Пример:** [1, a, “help”, True]

- Стойността на един контейнер включва стойностите на отделните му елементи
- Ако контейнерът е **immutable**, това означава че указателите на елементите не се променят. Но ако даден елемент е от тип **mutable**, стойността му може да се промени.

# Контейнери низове

- Всеки контейнер от тип низ включва в себе си произволен брой елементи, но само от тип символ

*Пример:* 'help'

- Стойността на един низ включва стойностите на отделните му елементи символи
- Контейнерът от тип низ е **immutable**, а това означава, че нито един негов елемент не може да се промени. Всеки елемент от един низ е символ (специален низ с един елемент), който също е **immutable**, стойността му не може да се промени.
- Контейнерите в общия случай съдържат елементи от произволен тип данни

# Вградени типове

Типове	Примери за стойности
Числа	1234, 3.1415, 3+4j
Низове	'spam', "guido's"
Логически	True, False
Списъци	[1, [2, 'three'], 4]
Речници	{'food': 'spam', 'taste': 'yum'}
Редици	(1,'spam', 4, 'U')
Файлове	myfile = open('eggs', 'r')

# Списъци

- ✓ Елементите се задават в квадратни скоби, отделени с ','
- ✓ Могат да имат елементи от различен тип данни
  - `a = ['spam', 'eggs', 100, 1234, 2*2]`
- ✓ Има достъп до всеки елемент или под-списък:
  - `a[0] → spam`
  - `a[:2] → ['spam', 'eggs']`
- ✓ Списъците могат да се променят (за разлика от низовете)
  - `a[2] = a[2] + 23`
  - `a[2:] = [123, 1234, 4]`
  - `a[0:0] = []`
  - `len(a) → 5`



## Списъци 2

- ✓ Могат да бъдат с произволна и променлива дължина, която може да се разбере с вградената функция `len()`
- ✓ Всеки елемент на списъка може да бъде обект от произволен тип данни, включително контейнер, и в частност списък
- ✓ Всеки обект от тип списък (`list`) е `mutable` - може да се променя (за разлика от низовете)
- ✓ Списъците наподобяват масивите в математиката но за разлика от тях допускат като елементи обекти от произволен тип (масивите съдържат обекти от един тип данни)

# Оператори за списъци

Допустими са същите както и за низове, имат на практика същото действие:

+ - слепване (конкатениране)

\* - размножаване (копиране)

in - проверка за наличие на елемент

Примери:

```
>>> [1, 2, 3] + [4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

```
>>> ['Ni!'] * 4
```

```
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

# Индексиране на списъци

- ✓ Начинът на индексиране и достъп до елементи или под-списъци е същият както при низовете
- ✓ Всеки елемент на списък може да се променя (за разлика от низовете)
- ✓  $L[\text{first}:\text{last}:\text{step}]$  # всяко first, last, step може да го няма
- ✓ По премълчаване: first=0, last=-1, step=1
- ✓  $L[0]$  е първи елемент, а  $L[-1]$  е последният елемент
- ✓  $L[:] \rightarrow L[0:-1]$  # еквивалентно на L
- ✓  $L[::-1] \rightarrow$  еквивалентно на обърнат списък

# Многомерни списъци

- ✓ Достъпът до много мерни списъци е както при многомерните масиви в математиката:
- ✓  $L[i][j]$  – в двумерния списък  $L$  обозначава  $(j+1)$ -я елемент в списък, който е  $(i+1)$ -и елемент в  $L$
- ✓ Аналогично се обобщава за произволна размерност
- ✓  $L = [ [ [1,2,3], 4, 5, 6], [7, 8, 9] ]$
- ✓  $L[1][2] \rightarrow 9$  ;  $L[0][1] \rightarrow 4$  ;  $L[0][0][1] \rightarrow 2$
- ✓  $L[i][j][k]$  – задава  $k+1$ -и елемент в списък, който е вложен  $j+1$ -и елемент в друг списък, който е вложен  $i+1$ -и елемент в списъка  $L$



# Вградени функции за списъци

- ✓ Допустими са същите както и за низове, имат на практика същото действие (len, in, del, enumerate ...)
- ✓ Функцията list се използва за преобразуване на обекти от произволен тип в списъци

```
>>> print(list('spam'))
```

```
['s', 'p', 'a', 'm']
```

```
>>> print(list(range(-4,4)))
```

```
[-4, -3, -2, -1, 0, 1, 2, 3]
```

- ✓ Функцията range(first, last, step) връща обект итератор съдържащ числата first .. last през step



# Вградени методи-функции за списъци

<code>L.append(4)</code>	# добавя елемента 4 в края на L
<code>L.extend([5,6,7])</code>	# добавя елементи в края
<code>L.insert(i, X)</code>	# вмъква елемент в позиция i
<code>L.index(X)</code>	# връща позиция на първото # срещане на X в L
<code>L.count(X)</code>	# връща броят на срещанията # на X в L

# Примери 1

```
>>> L = [1, 2, 3]
```

```
>>> L.append(4)
```

```
>>> L
```

```
[1, 2, 3, 4]
```

```
>>> L.extend([5, 6, 7])
```

```
>>> L
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
>>> L.insert(3,3)
```

```
>>> L
```

```
[1, 2, 3, 3, 4, 5, 6, 7]
```

```
>>> L.index(5)
```

```
5
```

```
>>> L.count(3)
```

```
2
```

# Вградени методи-функции за списъци

L.sort()	# сортиране
L.reverse()	# обръщане
L.copy()	# копиране
L.clear()	# изтрива всички елементи
L.pop(i)	# изтрива i+1-я елемент и го # връща като стойност (прем.: -1)
L.remove(X)	# изтрива първото срещане на X в L

## Пример 2

```
>>> M=L.sort()
```

```
>>> L, M
```

```
([1, 2, 3, 3, 4, 5, 6, 7], None)
```

```
>>> L.reverse()
```

```
>>> L
```

```
[7, 6, 5, 4, 3, 3, 2, 1]
```

```
>>> M = L.copy()
```

```
>>> M
```

```
[7, 6, 5, 4, 3, 2, 1, 0]
```

```
>>> M = L.copy()
```

```
>>> M
```

```
[7, 6, 5, 4, 3, 2, 1, 0]
```

## Пример 3

```
>>> M
```

```
[7, 6, 5, 4, 3, 2, 1, 0]
```

```
>>> print(M.pop(3))
```

```
4
```

```
>>> M
```

```
[7, 6, 5, 3, 2, 1, 0]
```

```
>>> M.pop()
```

```
0
```

```
>>> M.remove(5)
```

```
>>> M
```

```
[7, 6, 3, 2, 1]
```



## Пример 4

```
>>> L = [1, 2, 3, 4, 5]
```

```
>>> id(L)
```

```
47768800
```

```
>>> id(L[1:4])
```

```
47767760
```

```
>>> L[1:4]
```

```
[2, 3, 4]
```

```
>>> L[1:4] = [4, 3, 2]
```

```
>>> L
```

```
[1, 4, 3, 2, 5]
```

```
>>>
```

# Редици (tuples)

- ✓ Могат да имат елементи от различен тип данни
  - `a = ('spam', 'eggs', 100, 1234, 2*2)`
  - `b = 2018, "Year", 3`
- ✓ Наподобяват списъци, но се задават в кръгли скоби
- ✓ Има достъп до всеки елемент или под-редица:
  - `a[0] → spam`
  - `a[:2] → ['spam', 'eggs']`
- ✓ Начинът на индексване и достъп до елементи или под-редици е същият както при низовете и списъците
- ✓ Редиците не могат да се променят (както низовете, и за разлика от списъците)

## Редици 2

- ✓ Могат да бъдат с произволна но фиксирана дължина, която може да се разбере с вградената функция `len()`
- ✓ Всеки елемент на редицата може да бъде обект от произволен тип данни, включително контейнер, и в частност редица (наричаме го под-редица)
- ✓ Всеки обект от тип редица (`tuple`) е `immutable` – затова дължината на всеки обект не може да се мени
- ✓ Всяка редица се съхранява в паметта като многомерен масив от адреси на съответните елементи – обекти (както списъците)

# Оператори за редици

Допустими са същите както за низове и списъци, имат на практика същото действие:

- + - слепване (конкатениране)
- \* - размножаване (копиране)
- in - проверка за наличие на елемент

Примери:

```
>>> (1, 2, 3) + (4, 5, 6)
```

```
(1, 2, 3, 4, 5, 6)
```

```
>>> ('Hi', ', ', ') * 3
```

```
('Hi', ', ', ', 'Hi', ', ', ', 'Hi', ', ', '')
```



# Създаване на редици

Празна редица се създава като: `()`

```
>>> empty = ()
```

```
>>> type(empty)
```

```
<class 'tuple'>
```

Редица от един елемент се задава като се постави след него запетайка: `t = 3,`

Присвояването `t=(<обект>)` присвоява на `t` директно обекта `<обект>`, а не създава редица от 1 елемент!

```
>>> s = (4)
```

```
>>> type(s)
```

```
<class 'int'>
```



# Създаване на редици 2

## Пакетирано създаване

```
>>> point1 = 2, 3
```

```
>>> point2 = 3, 2
```

```
>>> coord = point1, point2 # ((2, 3), (3, 2))
```

## Автоматично де-пакетиране

```
>>> x, y = coord          # x = (2,3) ; y = (3, 2)
```

```
>>> a1, a2 = x             # a1 = 2 ; a2 = 3
```

# Разширено многозначно съпоставяне

```
>>>Var1, *Var2 = "extended"
```

Тук "extended" се преобразува в:

```
['e', 'x', 't', 'e', 'n', 'd', 'e', 'd']
```

След това Var1 се съпоставя с 'e' , а Var2 се съпоставя с останалото: ['x', 't', 'e', 'n', 'd', 'e', 'd']

```
>>> a, b
```

```
('e', ['x', 't', 'e', 'n', 'd', 'e', 'd'])
```

# Разширено многозначно съпоставяне

```
>>> Var1, Var2 = ('e', ['x', 't', 'e', 'n', 'd', 'e', 'd'])
```

```
>>> *Var3, Var4 = Var2
```

Тук Var4 първо се съпоставя с 'd' , а после Var3 се съпоставя с останалото: ['x', 't', 'e', 'n', 'd', 'e']

```
>>> Var3, Var4  
(['x', 't', 'e', 'n', 'd', 'e'], 'd')
```

# Разширено многозначно съпоставяне

```
>>> Var3, Var4  
(['x', 't', 'e', 'n', 'd', 'e'], 'd')
```

```
>>> Var5, *Var6, Var7 = Var3
```

Тук Var5 първо се съпоставя с 'x' , после Var7 се съпоставя с 'e', и накрая Var6 се съпоставя с останалото: ['t', 'e', 'n', 'd',,]

```
>>> Var5, Var6, Var7  
('x', ['t', 'e', 'n', 'd'], 'e')
```

# Разширено многозначно съпоставяне

\* пред името се използва да укаже, че името ще се съпостави с 0 / произволен брой обекти

Прилича на операция под – низ (slice)

Основна разлика: \* връща *списък* с обекти

Операцията slice се прилага не само за низове, а за произволен тип контейнер. Тя връща резултат от същия тип контейнер.



# Създаване на редици 3

```
>>> t = 123, [4,5,6], 'hello!'
```

```
>>> t[0]
```

```
123
```

```
>>> t
```

```
(123, [4,5,6], 'hello!')
```

```
>>> u = t, (1, 2, 3, 4, 5)
```

```
>>> u
```

```
((123, [4,5,6], 'hello!'), (1, 2, 3, 4, 5))
```

```
>>> t[0] = 88
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

# Вградени функции за редици

Допустими са същите както за низове и списъци, имат същото действие (len, in, del, enumerate)

```
>>> type((1, 2, 3))
```

```
<class 'tuple'>
```

```
>>> T = tuple('String')
```

```
>>> T
```

```
('S', 't', 'r', 'i', 'n', 'g')
```

```
>>> T = tuple([1, 2, 3])
```

```
>>> T
```

```
(1, 2, 3)
```

# Вградени функции за редици

```
>>> t
```

```
(2, True, 'example')
```

```
>>> t.count(2)
```

```
1
```

```
>>> t.count(22)
```

```
0
```

```
>>> t.index(True)
```

```
1
```

```
>>> t1 = 4, 8, 2, 6, 3
```

```
>>> sorted(t1)
```

```
[2, 3, 4, 6, 8]
```

## Пример с вградена функция zip()

```
>>> first_names = ["Петя", "Крис", "Мони"]  
>>> last_names = ["Иванова", "Петрова", "Стоева"]  
>>> Names = zip(first_names, last_names)  
>>> Names  
<zip object at 0x00000000002E03740>  
>>> for i in Names: print(i)
```

('Петя', 'Иванова')

('Крис', 'Петрова')

('Мони', 'Стоева')

# Заключение

- Изразите като команди
- Приоритет на операторите
- Многозначно и кратко присвояване
- Разклоняване на изпълнението с условна команда
- Условен оператор за изпълнение
- Списъци
- Оператори и функции за работа със списъци
- Редици
- Оператори и функции за работа с редици