Множество (Set)

- ✓ Неупорядоченный набор без повторяющихся элементов
- ✓ Имеются математические операции: объединение (union), пересечение (intersection), разность (difference) и симметричная разность

```
A = {1, 3}                  # создается set {1,3}
A.add(2)                    # 1,2,3
A.add(2)                    # 1,2,3 все элементы уникальны
B = set()                   # пустой set, а {} это пустой dictionary
B.update([6,5,4,3,4,5])     # {3, 4, 5, 6} обновление из list, tuple...
for i in B: print(i)        # цикл по set: 3,4,5,6
len(B)                      # 4 количество элементов
```

frozenset: замороженное, (не может быть изменено) множество

- ✓ `FB = frozenset(B)` `# FB такое же как B, но immutable`

two sets for the examples

```
A = {1, 2, 3, 4, 5}; B = {4, 5, 6, 7, 8}
```

операции с множествами

```
U = A.union(B)           # {1, 2, 3, 4, 5, 6, 7, 8} union of two sets
U = A | B                # the same with operator '|'
I = A.intersection(B)    # {4, 5} intersection of two sets
I = A & B                # the same this operator '&'
D = B.difference(A)      # {8, 6, 7} only in B but not in A
D = A - B                # {1, 2, 3} only in A but not in B
SD = A.symmetric_difference(B)  # in A and B but not in both
SD = A ^ B                      # {1, 2, 3, 6, 7, 8}
```

генераторы множеств (set comprehensions)

☞ аналогичны генераторам списков, но заключены в фигурные скобки

```
A= {x for x in 'This is a set comprehension' if x not in 'abracadabra'}
print(A) # {' ', 'p', 'h', 'T', 'i', 'm', 'o', 'n', 'e', 's', 't'}
```

Ассоциативный массив (Dictionary)

- ✓ Ассоциативный массив задает соответствие между множеством ключей `keys` и множеством значений `values`
- ✓ содержит пары (k, v) где k ключ, а v значение
- ✓ ключ `key` уникален: может встречаться в словаре только один раз

Example: name of coordinate → value

```
pp={}                                    # empty dictionary
pp={'x':1.2,'y':-4.3,'z':456}            # 3 entries
pp['r']=sqrt(sum([v**2 for v  in p]))    # add one more entry
pp['theta']=acos(pp['z']/pp['r'])        # add one more entry
pp['phi']=atan2(pp['y'],pp['x'])         # add one more entry
rho=pp['rho']                            # KeyError!
rho=pp['r']                              # access
del pp['x']                              # remove key-value pair
```

функция get(key)

```
print(pp.get('y'))              # -4.3
print(pp.get('rho'))            # None функция "ничего не вернула"
if pp.get('rho') == None:
   print('no rho')              # no rho
```

в словаре цикл for идет по всем ключам

```
for k in pp:
    print(k,'->',pp[k])
```

генераторы словарей (dict comprehensions)

☞ аналогичны генераторам множеств, но в левой части перед for стоит пара ключ : значение разделенные двоеточием

```
C = {x : int(x,16) for x in 'ABCDEF'}
print(C) # {'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15}
```

Оператор in

- **val in object:**
  - ✓ для `list`, `tuple`, `string`, `set` проверяет наличие указанного значения в наборе
  - ✓ для словаря проверяет наличие указанного ключа

Пример: list

```
l=['white','black','red','blue']
'red' in l      # True
'Red' in l      # False
'Red' not in l  # True
```

Пример: dictionary

```
d={1:'o', 2:'t', 10:'x'}
1 in d          # True
'o' in d        # False
10 not in d     # False
```

☞ `val not in obj` ≡ `not(val in obj)`

Функции

- Создание функции:
  ```
  def func_name(список параметров функции):
      operators
  ```
- Вызов:
  ```
  func_name(аргументы функции)
  ```

☞ параметры – имена в определении функции

☞ аргументы – значения фактически передаваемые в функцию

- ✓ Различают функции: глобальные, вложенные, лямбда-функции и методы
- ✓ Функция всегда возвращает значение: если нет `return` (или `return` без значения) то возвращается специальное значение `None`
- ✓ Возвращаемое значение можно игнорировать
- ✓ Возможен рекуррентный вызов как в C

Аргументы функции: то что передается в функцию

1. **«Позиционные» аргументы (positional arguments):**
   - ✓ значение аргумента при вызове определяется порядком следования

2. **«Именованные» аргументы (keyword arguments):**
   - ✓ значение аргумента определяется «именем ключа» (параметра)
   - ☞ располагаются после позиционных

тестовая функция

```
def fun(a,b,c):
    return a+b+c
```

```
# call fun() with positional args
fun(1,2,3)              # 6
fun('a','b','c')        # 'abc'
fun([1,2],['a'],[])     # [1, 2, 'a']
```

```
# call fun() with keyword args
fun(a='a',b='b',c='c')    # 'abc'
fun(c='a',b='b',a='c')    # 'cba'

#positional followed by key args
fun('a',c='b',b='c')      # 'acb'
```

3 Задание аргументов по умолчанию (default argument values)

- ✓ после параметра указывается значение «по умолчанию»: `name=default`
- ✓ при вызове соответствующие аргументы «необязательны»
- ☞ такие параметры располагаются после «обязательных»

Аргументы по умолчанию

тестовая функция

```
def fun(a,b='B',c='C'):
    return a+b+c
```

```
# positional arguments
fun('1')          # 1BC
fun('1','2')      # 12C
fun('1','2','3') # 123
```

```
# keyword arguments
fun(c=1,a=2,b=3)    # 6
fun(c='1',a='2')    # 2B1
fun(b='1',a='2')    # 21C
# positional followed by key args
fun('1',c='D')          # 1BD
```

☞ Специальные параметры функции / и * (⩾ python3.8)

```
def fun(positional_only,/, standard, *,keywords_only ):
```

- ✓ позволяют ограничить способ передачи аргументов: до / могут стоять только позиционные, после * только именованные
- ✓ используют для улучшения читаемости и быстроты выполнения

Только позиционные аргументы

```
def pos_only_args(a,b=1,/):
   return a+b
```

```
pos_only_args(0)       # 1
pos_only_args(2,3)     # 5
pos_only_args(2,b=2)   # TypeError
```

Только именованные аргументы

```
def key_only_args(*,a,b=1):
   return a+b
```

```
key_only_args(a=0)      # 1
key_only_args(a=2,b=3) # 5
key_only_args(2,b=2)    # TypeError
```

4 «Переменное» (arbitrary) число позиционных аргументов

- ✓ параметр функции задают в виде: `*name`
- ✓ в функции `name` — имя кортежа содержащего набор аргументов
- ✓ идет после всех позиционных аргументов
- ✓ после `*name` могут идти только именованные аргументы

Простой пример

```
def fun(a,*b):
    print(a,b)
```

```
fun(1,2,3,'f')  # 1 (2, 3, 'f')
fun(1)          # 1 ()
```

Вычисление суммы $\sum_i x_i^{deg}$

```
def sum_of_degrees(*x,deg=2):
   return sum(v**deg for v in x)
```

```
sum_of_degrees(3,4)          # 25
sum_of_degrees(3,4,deg=1)    # 7
```

5 «Переменное» (arbitrary) число именованных аргументов

- ✓ параметр функции задается в виде: `**name`
- ✓ в функции `name` — имя словаря с парами `имя : значение`
- ✓ `**name` должен быть последним параметром

Простой пример

```
def fun(a='a',**dic):
    print(a,dic)
```

```
fun(1,rep=1,pop=2)       # 1 {'rep': 1, 'pop': 2}
fun(rep=1,pop=2,cop=3)   # a {'cop': 3, 'rep': 1, 'pop': 2}
fun()                    # a {}
```

Распаковка (unpacking) аргументов из «наборов данных»

- ✓ что бы передать содержимое списка/кортежа/set в функцию ожидающую набор аргументов используют оператор распаковки *
- ✓ для словаря используется оператор распаковки **

операторы распаковки * и **

тестовая функция

```
def fun(a,b,c):
    return a+b+c
```

```
t=(2,3,4)                # tuple
fun(*t)                  # 9
l=[2,3,4]                # list
fun(*l)                  # 9
d={'a':2,'b':3,'c':4}    # dict.
fun(**d)                 # 9
```

пример-2

тестовая функция

```
def sum_of_squares(*arg):
    return sum(v**2 for v in arg)
```

```
sum_of_squares(2,3,4)           # 29
sum_of_squares(*t)              # 29
sum_of_squares(*l)              # 29
sum_of_squares(*t[:2],*l[1:]) # 38
```

Изменяемые и неизменные объекты как аргументы функции

- ✓ аргументы в функцию передаются по ссылке (“pass by reference”)
- ✓ если объект неизменяемый (“immutable”), функция его изменить не может; изменяемые объекты (“mutable”) могут в функции меняться
  - **Неизменяемые:** `int, float, complex, tuple, string, frozenset`
  - **Изменяемые:** `list, set, dictionary, user-defined classes`

Test mutable/immutable objects

тестовая функция

```
def fun(x,y):
  print(x is y,end=';') #operator is
  x += y
  print(x is y,':',x)
```

список

```
L=[1,2]
fun(L,L)       #True;True : [1,2,1,2]
print('L=',L) #L= [1, 2, 1, 2]
```

число

```
i=1
fun(i,i)        #True;False : 2
print('i=',i)   #i= 1
```

текст

```
s='str'
fun(s,s)        #True;False : strstr
print('s=',s)   #s= str
```

кортеж

```
t=3,
fun(t,t)        #True;False : (3, 3)
print('t=',t)   #t= (3,)
```

Локальные и глобальные переменные в функции

Переменная внутри функции, которая не являются параметром:

- **Локальная:** если ей присваивается значение
- **Глобальная:** используется только на «чтение» или она объявлена как глобальная с помощью `global`

- **инструкция `global` сообщает, что переменная глобальная**

```
def fun(x):
    # print(y) - UnboundLocalError
    y=1  # y is local here
    return x+y
y=10
print(fun(1),y) # 2 10
```

```
def funG(x):
    global y
    y=1
    return x+y
y=10
print(funG(1),y) # 2 1
```

Вложенные функции (nested functions)

- ✓ в python функцию можно определить внутри другой функции
- ✓ все локальные переменные внешней функции доступны во вложенной функции только «для чтения»

```
def fun()
    i=1                          # 'i' is local in fun()
    def f2(n):                   # nested function
        return n*(n+i)//2        # use 'i' variable from fun()
    return f2(1)+f2(3)+f2(5)

print(fun(2))  # 25
```

☞ Использовать `f2()` можно только внутри `fun()`

- Служебное слово `nonlocal`
  - ✓ объявляет, что переменная не является локальной переменной вложенной функции

```
def fun()
    i=1  # 'i' is local in fun()
    def f4(n):
        nonlocal i   # declare that 'i' is not local in f4()
        i += 1       # use 'i' variable from fun()
        return n*(n+i)//2
    return f4(1)+f4(2)+f4(3)
print(fun(0))    # 14
```

Функции как объекты

- **В python функция это объект (first class object)**
  - ✓ можно использовать как аргумент другой функции
  - ✓ может быть возвращаемым значением другой функции
  - ✓ может содержаться в различных структурах, например в списке или словаре

Пример: функция как аргумент

тестовая функция

```
def operate(f,x):
    return f(x)
```

```
import math
operate(math.sqrt,math.pi)              # 1.7724538509055159
operate(sum_of_squares,2)               # 4
```

Лямбда-функции (анонимные функция)

Lambdas, lambda-function, анонимная (без имени) функция

- ✓ лямбда-функция создаются выражением: `lambda` *parameters*`:` `expression`
- ✓ параметры и аргументы такие же как у обычной функции
- ✓ операторы в `expression` не должны содержать условные инструкции и циклы
- ✓ обычно создается в месте использования: внутри обычных функций
- ✓ лямбда-функцию можно сохранить в переменную и тогда она приобретет имя

```
operate(lambda x: x**2, 4) # lambda function inside of operate()
sq=lambda x: x**2          # save lambdas to variable
operate(sq,4)              # 16

sum_sq=lambda *vec: sum(x**2 for x in vec) # vec: variable list of args
sum_sq(1,2,3)                              # 14
```

Использование в «ключевых функциях» (key functions)

- ✓ функция возвращающая значение по которому делается сортировка
- ✓ используется в: `sorted()`,`list.sort()`, `min()`, `max()` ...

Пример для `sorted()`

list of tuples we want to sort

```
lt = [(0,'black'), (1,'white'), (3,'red'),(2,'blue')]

sorted(lt,key=lambda pair: pair[0]) # sort by number
#  [(0, 'black'), (1, 'white'), (2, 'blue'), (3, 'red')]

sorted(lt,key=lambda pair: pair[1]) # sort by color name alphabetically
# [(0, 'black'), (2, 'blue'), (3, 'red'), (1, 'white')]

sorted(lt,key=lambda pair: len(pair[1])) # sort by length of color name
# [(3, 'red'), (2, 'blue'), (0, 'black'), (1, 'white')]
```

Несколько возвращаемых значений

- Вернуть или `tuple` или `list` или `dictionary`

using tuple

```
def sin_cos(alpha):
    return sin(alpha),cos(alpha)  # tuple with two elements

s,c = sin_cos(0.1)  # unpack tuple to variables s and c
```

`main()` функция

- в питоне нет «стартовой» точки, но часто используется прием со специальной переменной `__name__`

```
def main():
   # code for main()-function here

if __name__ == "__main__":
    main()
```