Python函数

函数：一段可以被重复使用的代码

- 利于代码重用减少冗余
- 将一个复杂的操作分解为独立子操作，代码逻辑清楚

matplotlib.pyplot.plot(*args, **kwargs)

Plot lines and/or markers to the Axes. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

创建和调用函数 ¶

```
def func_name(arg1, arg2,... argN):
    statements
    return value
```

C/C++和Fortran的函数在编译前就已经是确定的。而def是python的标准语句，可以在程序允许语句的地方出现。

```
if test:
    def func(x):
        return x += 1
else:
    def func(x):
        return x -= 1
func(1)
```

In [ ]:
```
def sum(x, y):
    return x + y
```

In [ ]:
```
sum(1, 2)
```

- 函数名指向一个函数对象

In [ ]:
```
sum_alias = sum
sum_alias(3, 4)
```

In [ ]:
```
def times(x, y):
    return x * y
sum = times
sum(1, 2)
```

- 模拟多变量返回（序列展开赋值）

In [ ]:
```
def func(x):
    return x + 1, x*10

a, b = func(2)
print(a, b)
```

变量作用域

Python语言中的变量名都有一个作用域（生存期）

- 所有在函数中创建的变量名的作用域都在函数语句块内
- 在def之外的变量名为整个python源文件

换而言之

- 一个python源程序文件定义了一个全局(global)域
- 每个函数定义了一个局部(local)域

Python名字查询规则（LEGB）：

- 首先搜索局部域(Local)
- 其次搜索包围域(Enclosing)
- 再次搜索全局域(Global)
- 最好搜索内置域(Build-in)

In [ ]:
```
X = 99

def func(Y):
    Z = X + Y
    return Z

func(1)
```

In [ ]:
```
X = 99

def func():
    X = 88

func()
print(X)
```

global和nonlocal语句

- global 在局部域中修改全局域变量名

- nonlocal 局部域中修改上一级包含域中的变量名

In [ ]:
```
X = 99

def func():
    global X
    X = 88

func()
print(X)
```

In [ ]:
```
X = 99

def f1():
    X = 88
    def f2():
        X = 77
    f2()
    print('X in f1', X)
f1()
print('X in global', X)
```

In [ ]:
```
X = 99

def func():
    nonlocal X
    X = 88

func()
```

函数参数传递

In [ ]:
```
Python函数参数传递要点
* 函数的参数名成为局部变量名
* 参数传递是对函数局部变量赋值的过程
* 在函数内部修改**可修改**类型的对象可能影响函数调用者
* **不可修改immutable**类型参数可等同为“传值”
* **可修改mutable**类型参数可等同为“传址”
```

In [ ]:
```
def func(a):
    a = 99

b = 88
func(b)
print(b)
```

In [ ]:
```
def changer(a, b):
    a = 2
    b[0] = 'str'

X = 1
L = [1, 2]
changer(X, L)
print(X, L)
```

Python函数的参数匹配

Python函数在**定义**时，参数列表可以有如下4种形式:

```
def func(name)              #普通参数
def func(name=value)        #默认值参数
def func(*name)             #任意位置参数，收集所有额外位置参数组创建一个tuple
def func(**name)            #任意关键字参数，收集所有额外关键字参数创建一个字典(Dict)
```

如果这四种参数列表同时存在，需要按如下的顺序排列:

```
def func(name, *extra_pos, default=value, **extra_keyword)
```

Python在**调用**函数时，参数列表也可以有如下4种形式:

```
func(name)                  #普通参数，按位置匹配
func(name=value)            #关键字参数，按名字匹配
func(*name)                 #将一个序列展开，作为多个普通参数传递
func(**name)                #将一个字典展开，作为多个关键字参数传递
```

这四种参数同时存在时，需要按如下的顺序排列:

```
func(name, keyword=value, *seq, **dicts)
```

函数定义仅包含普通参数

In [ ]:
```
def func(a, b, c):
    print(a, b, c)
```

In [ ]:
```
func(1, 3, 5)       # 按位置匹配
```

In [ ]:
```
func(c=5, a=1, b=3) # 按关键字匹配
```

In [ ]:
```
args = (1, 3, 5)
func(*args)         # 展开可迭代对象，并按位置赋值
```

In [ ]:
```
args = {'c':5, 'a':1, 'b':3}
func(**args)        # 展开字典，并按关键字匹配
```

函数定义包含普通参数和默认值参数

In [ ]:
```
def func(a, b, c=0):
    print(a, b, c)
```

In [ ]:
```
func(1, 3)          # 按位置匹配，c使用默认值
```

In [ ]:
```
func(1, 3, 5)       # 按位置匹配全部
```

In [ ]:
```
args=(1, 3, 5)
func(*args)         # 展开序列，并按位置匹配
```

In [ ]:
```
args={'b':3, 'a':1}
func(**args)        # 展开字典，并按名字匹配
```

函数定义包含任意位置和关键字参数

In [ ]:
```
def func(a, b, c=0, *args, **kargs):
    print(a, b, c, args, kargs)
```

In [ ]:
```
func(1,2)
```

In [ ]:
```
func(1, 3, 5)
```

In [ ]:
```
func(1, 3, 5, 6, 7)
```

In [ ]:
```
func(1, 3, 5, 6, 7, c=0, d=5)
```

In [ ]:
```
func(**{'a':1, 'b':2, 'c':3})
```

lambda函数

匿名函数: 创建之后不赋值给变量名，而是直接返回函数对象。

lambda函数基本形式：

```
lambda arg1, arg2,... argN : expression
```

- lambda只是一般表达式，不是一个语句，因此可以用到不允许语句的地方
- lambda的函数体只能是单个表达式

In [ ]:
```
def func(x, y, z): return x + y + z

func(1, 2, 4)
```

In [ ]:
```
f = lambda x, y, z : x + y + z
f(1, 2, 4)
```