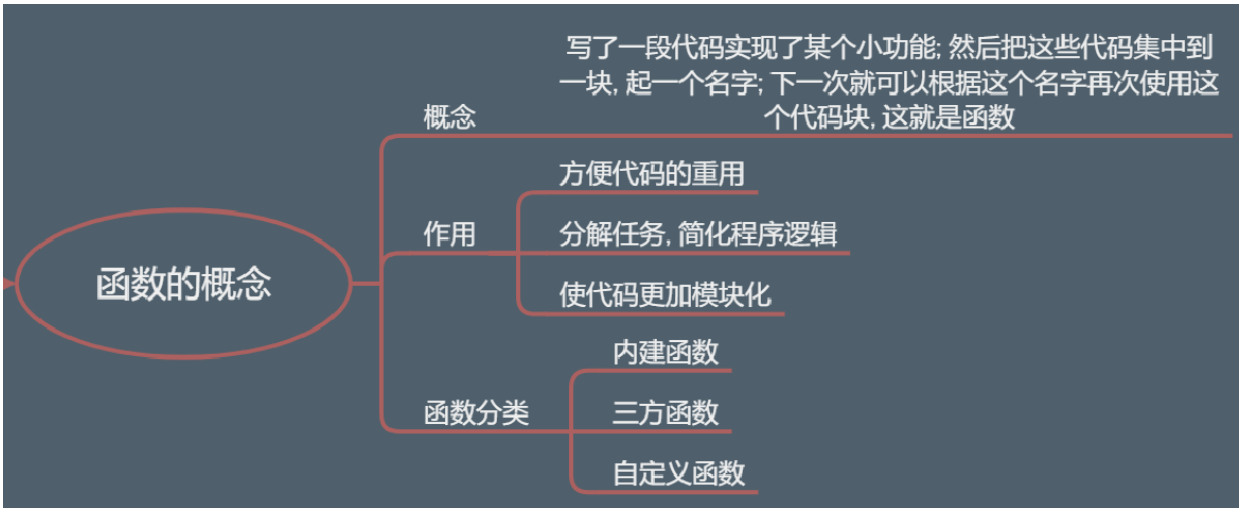
[笔记][LIKE-Python-3]

Python

002. Python函数-基本概念



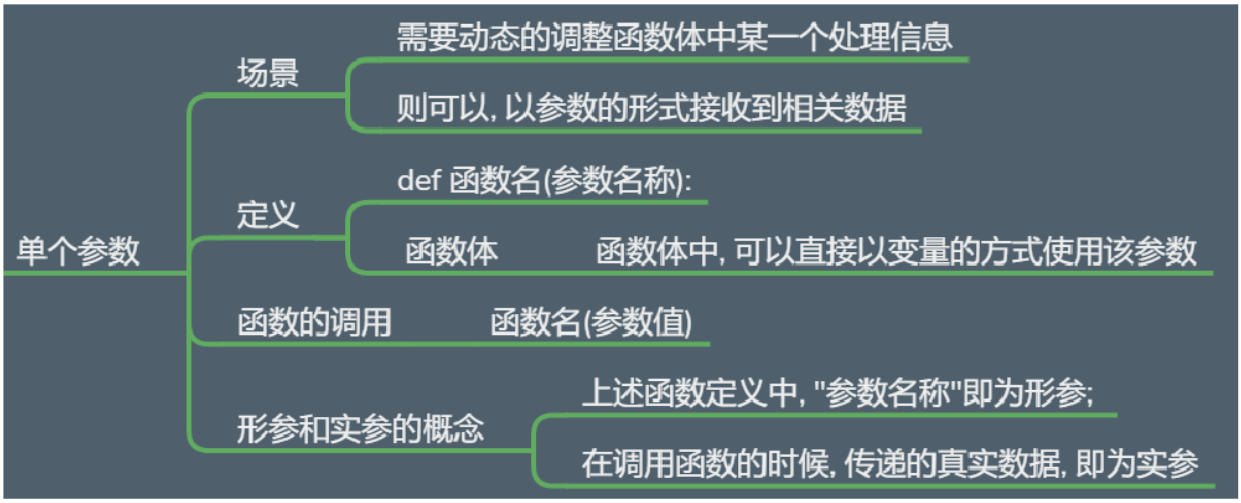

```
# 文件的大小会变大
# 代码的冗余较大，重用性比较差
# 代码的可维护性比较差

print(1)
print(12)
print(3)
print(14)
print(5)
print(16)
print(7)
print(18)
print(9)

def p_func():
    print(1)
    print(12)
    print(3)
    print(14)
    print(5)
    print(16)
    print(7)
    print(18)
    print(9)

p_func()
p_func()
```

003. Python函数-简单定义

> **函数**就是把在它后面有缩进的一段代码**包装起来**。

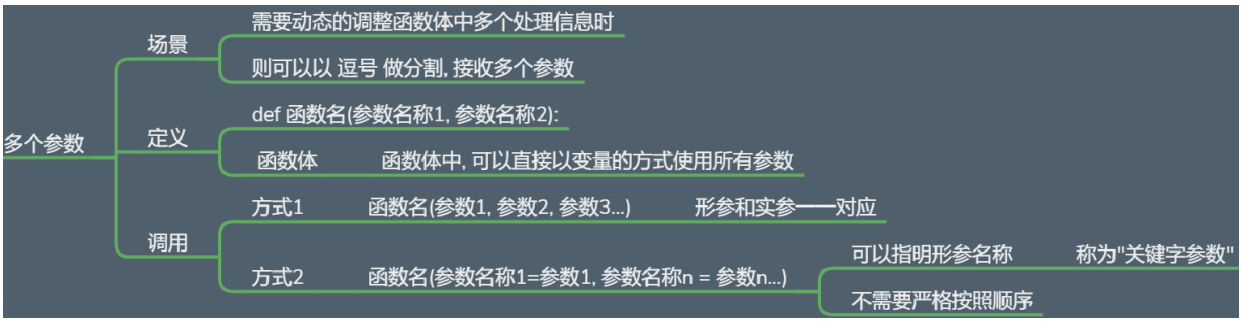
```
def test():
    print(2 ** 1)
    print(2 ** 2)
    print(2 ** 3)


test()
"""
2
4
8
"""
```

004. Python函数-参数-单一参数




```python
def test(num):
    print(num ** 1)
    print(num ** 2)
    print(num ** 3)


test(3)
"""
3
9
27
"""
```

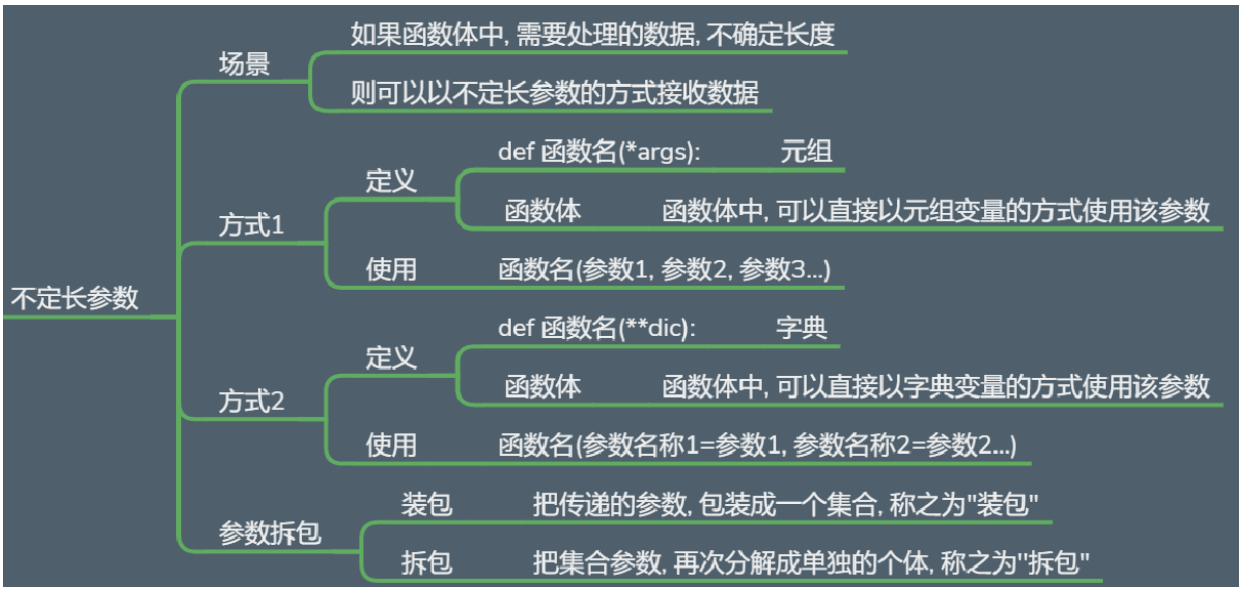
005. Python函数-参数-多个参数




```python
def my_sum(num1, num2):
    print(num1)
    print(num2)
    print(num1 + num2)

# 位置参数
my_sum(2, 4)
"""
```

```
2
4
6
"""

# 关键字参数
my_sum(num2=5, num1=6)
"""
6
5
11
"""
```

006. Python函数-参数-不定长参数-上



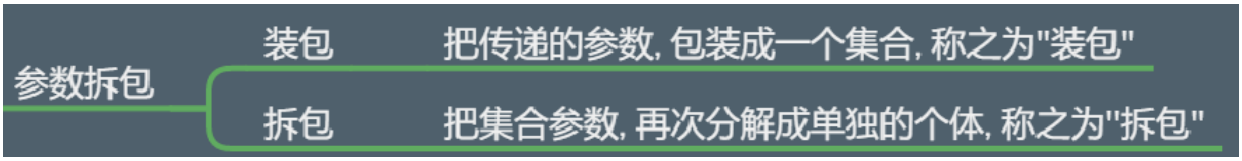

```
# 不定长参数
# 变长参数
# 可变长参数

def my_sum(*t):
    print(t, type(t))
    result = 0
    for v in t:
        print(v)
        result += v
    print(result)

my_sum(4, 5, 6, 7)
"""
(4, 5, 6, 7) <class 'tuple'>
```

```
4
5
6
7
22
"""
```

005. Python函数-参数-不定长参数-中

```
def my_sum(**kwargs):
    print(kwargs, type(kwargs))

my_sum(name='团子', age=12)
"""
{'name': '团子', 'age': 12} <class 'dict'>
"""
```

006. Python函数-参数-参数的拆包和装包



拆包又叫做解包

```
def my_sum(a, b, c, d):
    print(a + b + c + d)

def test(*args):
    print(args)
    # 拆包
    print(*args)
    my_sum(*args)

test(1, 2, 3, 4)
"""
(1, 2, 3, 4)
1 2 3 4
```

```
10
"""

def func(a, b):
    print(a)
    print(b)


def test2(**kwargs):
    print(kwargs)
    # 拆包
    func(**kwargs)

# 注意关键字参数的名字一定要对应
# test2(a=1, c=2) 会报错!
test2(a=1, b=2)
"""
{'a': 1, 'b': 2}
1
2
"""
```

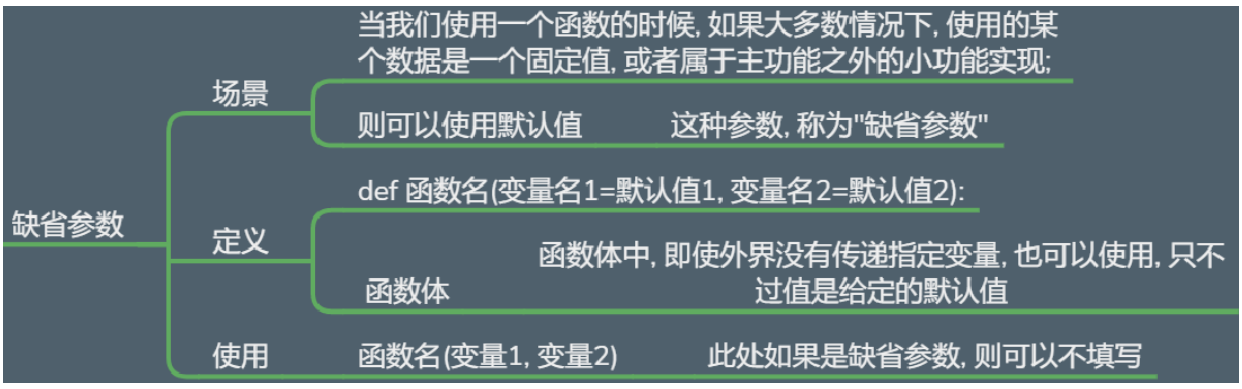
注意关键字参数的名字**一定要对应**。

> 【注】
> 关键字参数不能重复赋值

---

009. Python函数-参数-不定长参数-缺省参数

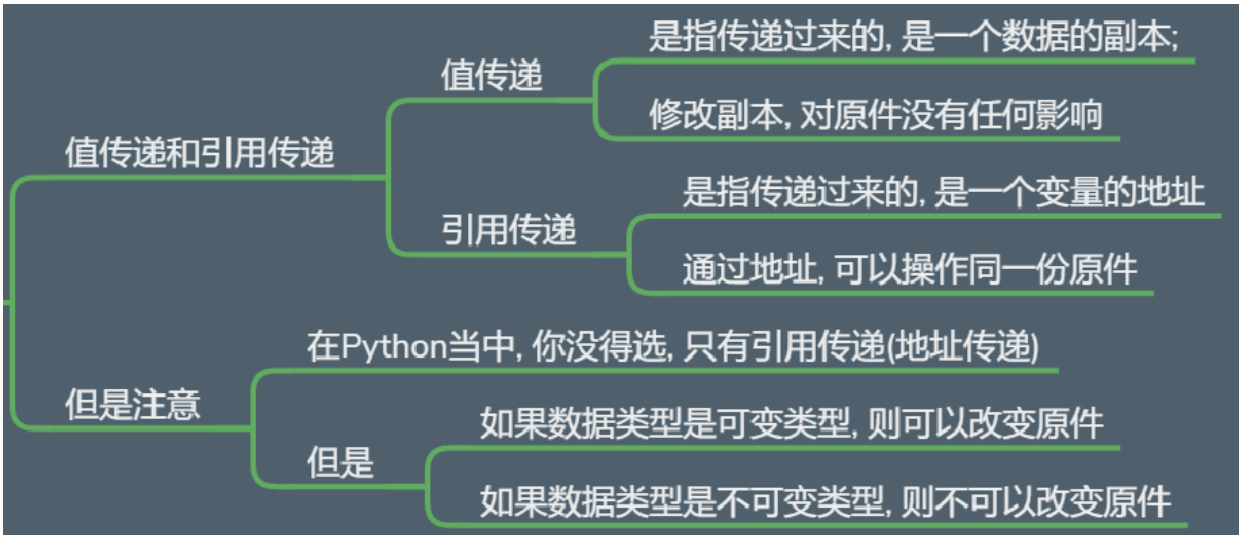
缺省参数又叫做**默认参数**。




```
result = sorted([1, 3, 2, 5, 4], reverse=True)
```

```
print(result)
"""
[5, 4, 3, 2, 1]
"""

def hit(somebody='豆豆'):
    print('我想打', somebody)
hit()
"""
我想打 豆豆
"""

hit('毛毛')
"""
我想打 毛毛
"""
```

010. Python函数-参数-函数的注意事项




```
def change(num):
    print('id为: ', id(num))
    print(num)

b = 10
print('id为: ', id(b))
change(b)
"""
id为:  1984064832
id为:  1984064832
10
"""
```

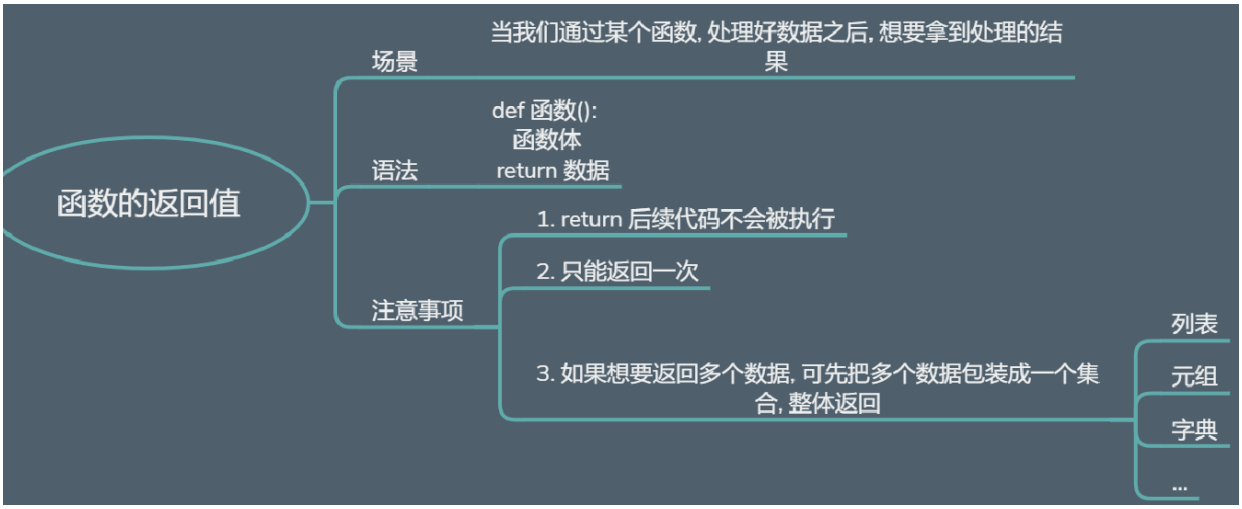
数字是不可变类型，作为参数在函数内不能改变。

```
def change(num):
    num = 666
    print('num的id为：', id(num))
    print('num的值为：', num)

b = 10
change(b)
print('b的id为：', id(b))
print('b的值为：', b)
"""
num的id为：  2763172390000
num的值为：  666
b的id为：  1984064832
b的值为：  10
"""
```

可变类型的参数可以在函数内改变。

```
def change(num):
    print('改变num之前的id为：', id(num))
    num.append(666)
    print('改变num之后的id为：', id(num))
    print('num的值为：', num)

b = [1, 2, 3]
change(b)
print('b的id为：', id(b))
print('b的值为：', b)
"""
改变num之前的id为：  2437526194376
改变num之后的id为：  2437526194376
num的值为：  [1, 2, 3, 666]
b的id为：  2437526194376
b的值为：  [1, 2, 3, 666]
"""
```

011. Python-函数-返回值

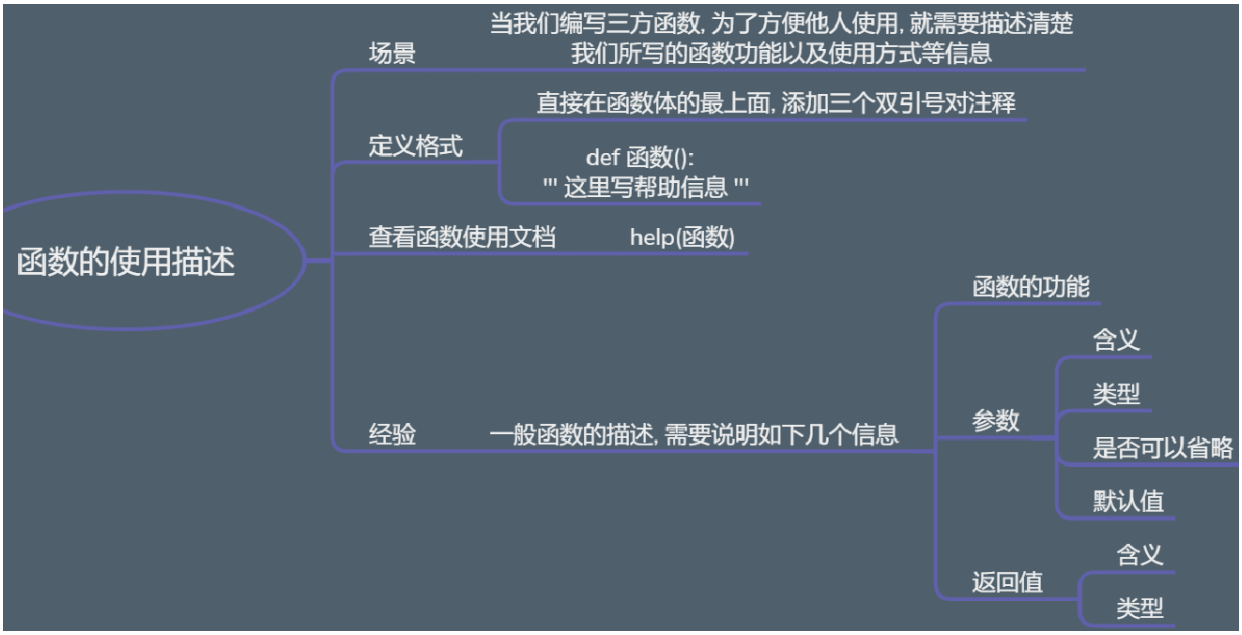



```python
def my_sum(a, b):
    result = a + b
    return result

res = my_sum(6, 7)
print(res)
"""
13
"""
```

【注】
`return` 返回 `None`
不写默认也返回 `None`

```python
def calc(a, b):
    he = a + b
    cha = a - b
    return he, cha

he, cha = calc(1, 2)
print(he)
print(cha)
"""
3
-1
"""
```

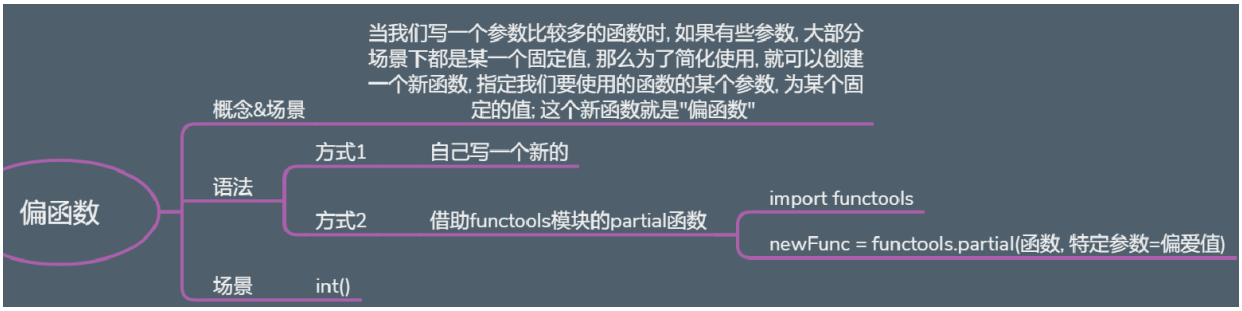
012. Python函数-函数的使用描述



【注】
文档字符串应该使用**三个双引号对**。

```
def calculate(a, b=1):
    """
    计算两数之和与两数之差。
    :param a: 数值1，数值类型，不可选，没有默认值
    :param b: 数值2，数值类型，可选，默认值：1
    :return: 返回计算结果，元组：(和，差)
    """
    return (a + b, a - b)

print(help(calculate))
"""
Help on function calculate in module __main__:

calculate(a, b=1)
    计算两数之和与两数之差。
    :param a: 数值1，数值类型，不可选，没有默认值
    :param b: 数值2，数值类型，可选，默认值：1
    :return: 返回计算结果，元组：(和，差)

None
"""
```

013. Python函数-偏函数



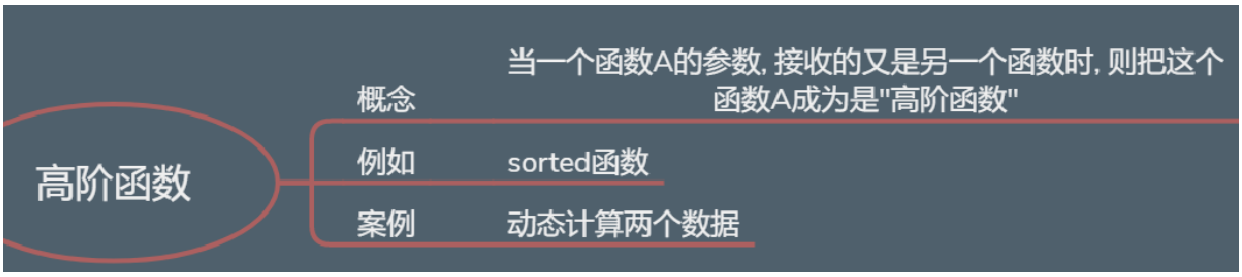

```python
def test(a, b, c, d=1):
    print(a + b + c + d)

def test2(a, b, c, d=2):
    test(a, b, c, d)

test2(1, 2, 3)
"""
8
"""

from functools import partial
# c 偏爱 5
new_func = partial(test, c=5)
print(new_func, type(new_func))
"""
functools.partial(<function test at 0x0000018054D81E18>, c=5) <class 'fun
ctools.partial'>
"""

new_func(1, 2)
"""
9
"""
```

014. Python函数-偏函数-使用场景

```python
num_string = '100010'
result = int(num_string, base=2)
print(result)
"""
34
"""

# 在往后的一段时间里
# 我都需要把一个二进制的字符串，转换成对应的十进制数据
from functools import partial
```

```
int2 = partial(int, base=2)
result = int2(num_string)
print(result)
"""
34
"""
```

015. Python函数-高阶函数



函数其实也是一个变量。
定义函数的时候，会在内存中开辟一块空间，然后把函数体放在里面。
让函数名引用这块空间。

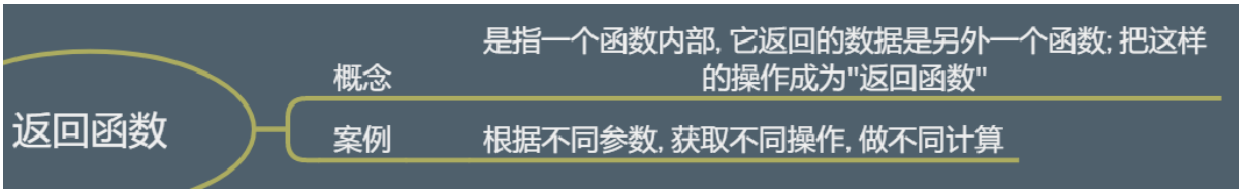
```
# a, b 是形参, 也就是变量
# 传递数据就是指给变量赋值
def test(a, b):
    print(a + b)

print(test)
print(id(test))
"""
<function test at 0x000001F982C11E18>
2171152178712
"""

# 函数本身也可以作为数据
# 传递给另外一个变量
test2 = test
test2(1, 2)
print(id(test2))
"""
3
2171152178712
"""

# 高阶函数: 接收函数作为参数的函数
# sorted
li = [
    {'name': '团子', 'age': 18},
```

```
    {'name': '小明', 'age': 20},
    {'name': '大锤', 'age': 19}]

# result = sorted(li)
# print(result)
"""
TypeError: '<' not supported between instances of 'dict' and 'dict'
"""

def get_key(x):
    return x['age']

result = sorted(li, key=get_key)
print(result)
"""
[{'name': '团子', 'age': 18}, {'name': '大锤', 'age': 19}, {'name': '小明',
 'age': 20}]
"""

# sorted 接收 get_key 作为参数，所以它是高阶函数
```

016. Python函数-高阶函数-使用场景

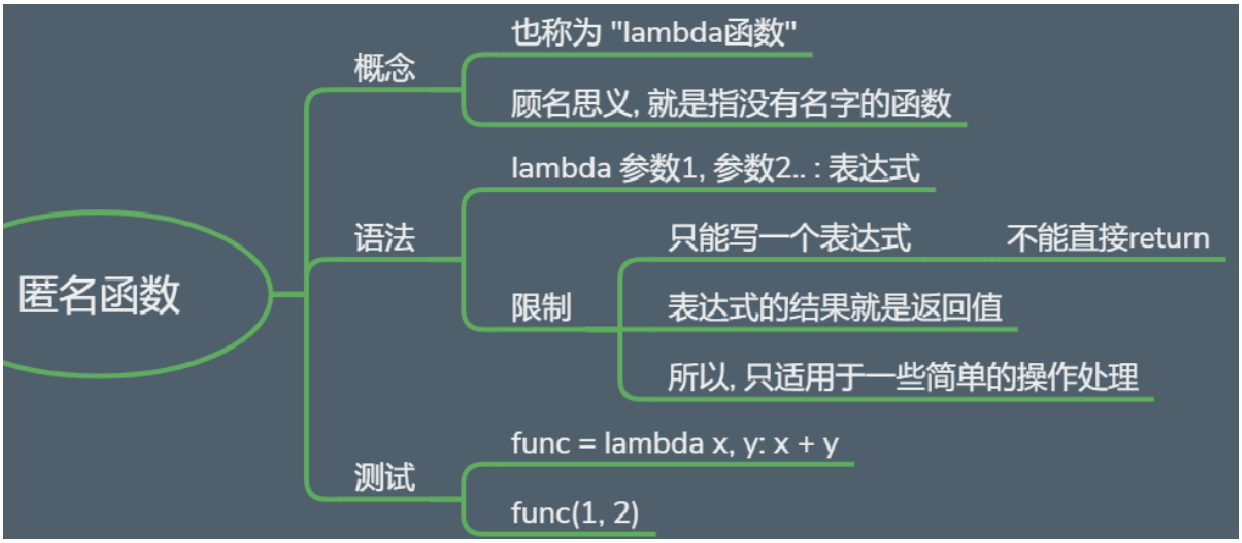
```
def calculate(num1, num2, calc_func):
    print(calc_func(num1, num2))

def add(a, b):
    return a + b

def sub(a, b):
    return a - b

calculate(1, 2, add)
calculate(1, 2, sub)
"""
3
-1
"""
```

017. Python函数-返回函数



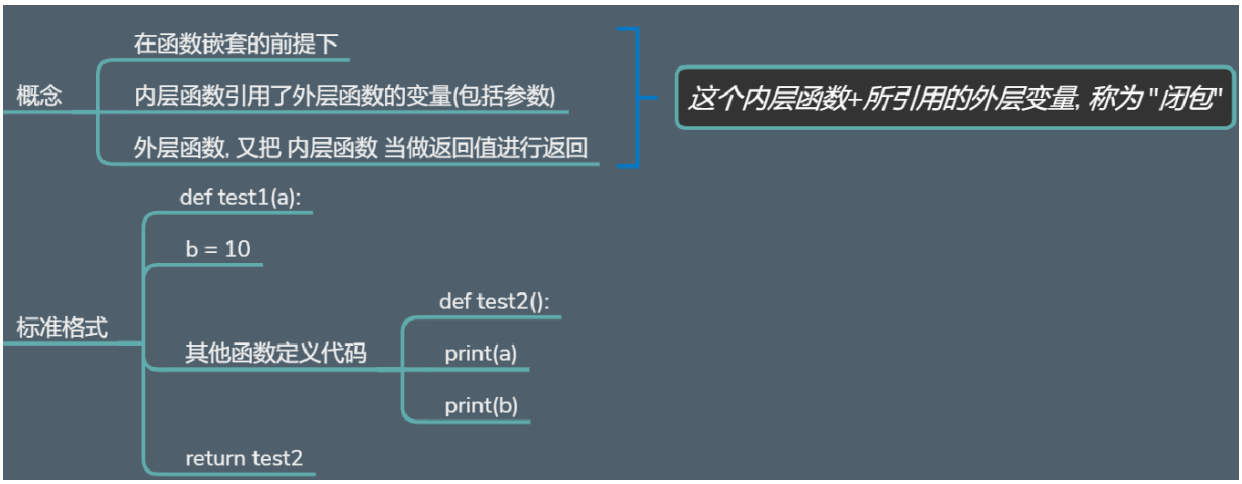

```python
# 返回函数
def get_func(flag):
    # 1. 再定义几个函数
    def add(a, b):
        return a + b

    def sub(a, b):
        return a - b

    # 2. 根据不同的 flag，返回不同的函数
    if flag == '+':
        return add
    elif flag == '-':
        return sub

result = get_func('+')
print(result, type(result))
"""
<function get_func.<locals>.add at 0x0000024D98AAFB70> <class 'function'>
"""

res = result(1, 7)
print(res)
"""
8
"""
```

018. Python函数-匿名函数(lambda函数)



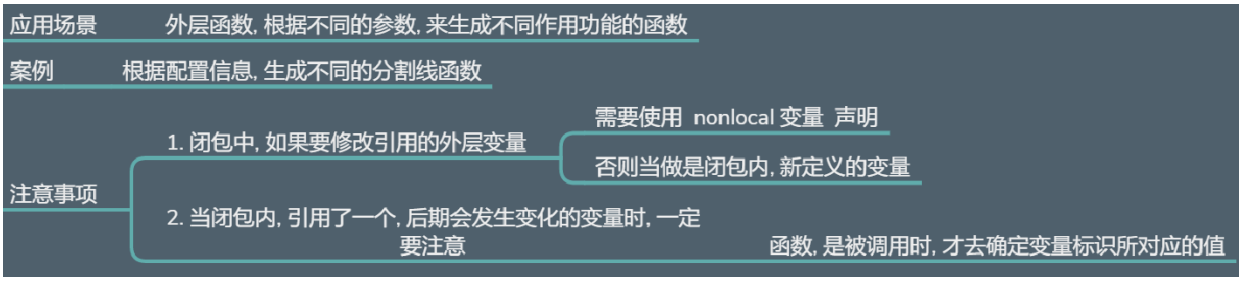

```python
# 匿名函数
result = (lambda x, y: x + y)(1, 2)
print(result)
"""
3
"""

func = lambda x, y: x + y
print(func(4, 5))
"""
9
"""

# 应用场景
li = [
    {'name': '团子', 'age': 18},
    {'name': '小明', 'age': 20},
    {'name': '大锤', 'age': 19}]
result = sorted(li, key=lambda x: x['age'])
print(result)
"""
[{'name': '团子', 'age': 18}, {'name': '大锤', 'age': 19}, {'name': '小明',
 'age': 20}]
"""

# 更好的写法
from operator import itemgetter
result = sorted(li, key=itemgetter('age'))
print(result)
"""
[{'name': '团子', 'age': 18}, {'name': '大锤', 'age': 19}, {'name': '小明',
 'age': 20}]
"""
```

019. Python函数-闭包-概念格式




```
def test():
    a = 2
    def test2():
        print(a)
    return test2

new_func = test()
new_func()
"""
2
"""
```

020. Python函数-闭包-小案例




```
def line_config(content, length):
    left = length // 2
    right = length - left
    def line():
        print('-' * left + content + '-' * right)
    return line

line1 = line_config('闭包', 30)
```

```
line1()
line2 = line_config('xxx', 20)
line2()
"""
---------------闭包---------------
----------xxx----------
"""
```

021. Python函数-闭包-注意事项-1

```
def test():
    num = 10
    def test2():
        num = 666
        print(num)
    print(num)
    test2()
    print(num)
    return test2

result = test()
"""
10
666
10
"""

def test():
    num = 10
    def test2():
        nonlocal num
        num = 666
        print(num)
    print(num)
    test2()
    print(num)
    return test2

result = test()
"""
10
666
666
"""
```

022. Python函数-闭包-注意事项-2

```
def test():
    a = 1
    def test2():
        print(a)
    a = 2
    return test2

new_func = test()
new_func()
"""
2
"""

# 并不会报错
# 因为 b 不是在定义的时候设置值的
def test():
    print(b)
print('说点什么')
"""
说点什么
"""
# 在调用函数的时候，才会去查找变量对应的值具体是什么
# test()
"""
NameError: name 'b' is not defined
"""

def test():
    funcs = []
    for i in range(1, 4):
        def test2():
            print(i)
        funcs.append(test2)
    return funcs

newfuncs = test()
print(newfuncs)

newfuncs[0]()
newfuncs[1]()
newfuncs[2]()
"""
[<function test.<locals>.test2 at 0x000001A6C95360D0>, <function test.<lo
cals>.test2 at 0x000001A6C95361E0>, <function test.<locals>.test2 at 0x00
0001A6C9536268>]
3
```

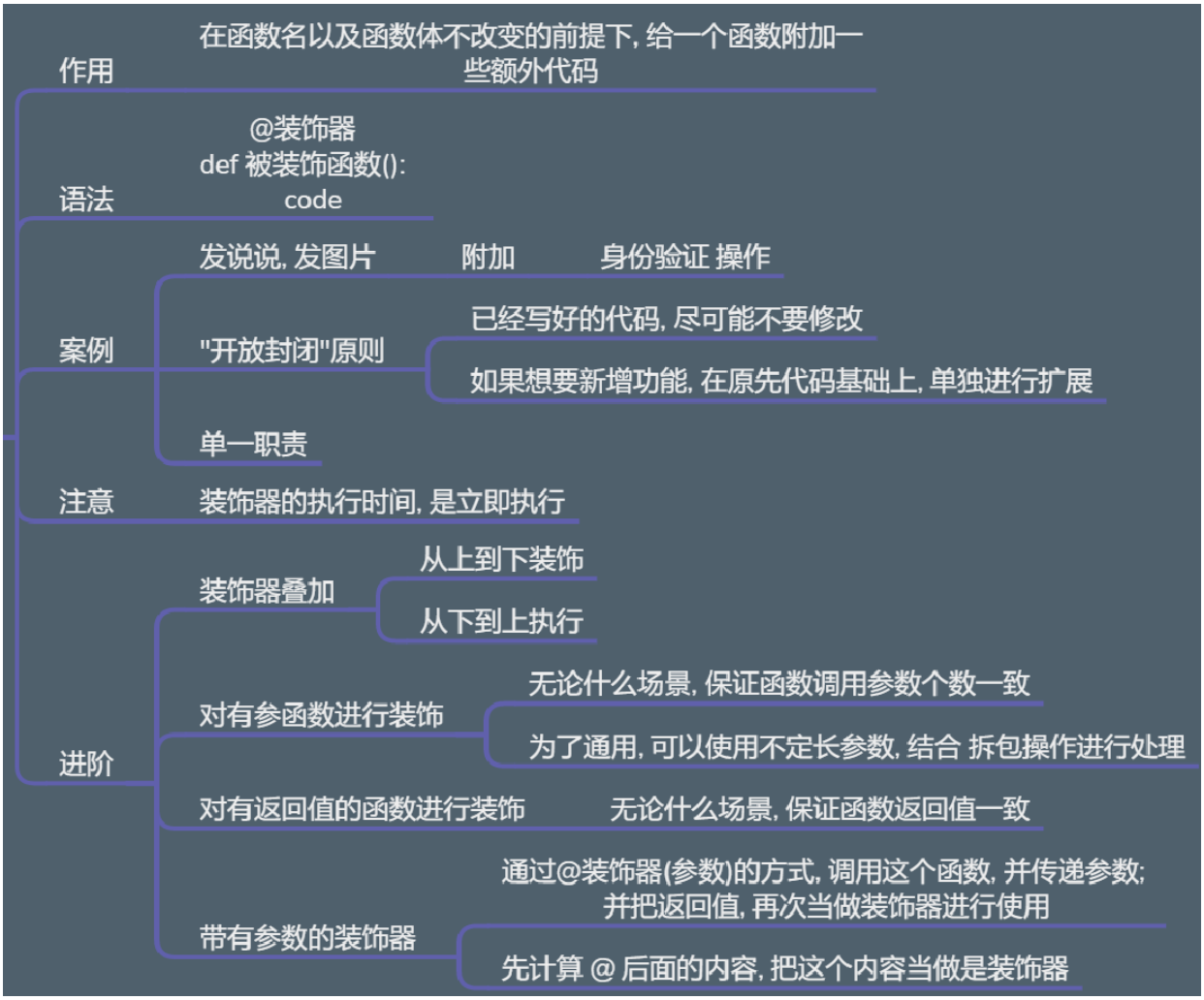
```
3
3
"""

# 解决方案
def test():
    funcs = []
    for i in range(1, 4):
        def test2(num):
            def inner():
                print(num)
            return inner
        funcs.append(test2(i))
    return funcs

newfuncs = test()
print(newfuncs)

newfuncs[0]()
newfuncs[1]()
newfuncs[2]()
"""
[<function test.<locals>.test2.<locals>.inner at 0x0000021F79EA6378>, <fu
nction test.<locals>.test2.<locals>.inner at 0x0000021F79CD1E18>, <functi
on test.<locals>.test2.<locals>.inner at 0x0000021F79EA6400>]
1
2
3
"""
```

闭包就是函数本身以及它引用的上下文环境。

---

023. Python函数-装饰器-案例-1

装饰器也是一种设计模式。




```python
# 定义两个功能函数
def send_shuoshuo():
    print('发说说')

def send_pic():
    print('发图片')

# 相关的逻辑代码
btn_index = 1
if btn_index == 1:
    send_shuoshuo()
elif btn_index == 2:
    send_pic()
```

功能函数和业务逻辑代码分开！

---

024. Python函数-装饰器-案例-2

```python
# 定义两个功能函数
def send_shuoshuo():
    print('发说说')

def send_pic():
    print('发图片')

# 相关的逻辑代码
btn_index = 1
if btn_index == 1:
    send_shuoshuo()
elif btn_index == 2:
    send_pic()

# 发说说，发图片的前提：必须要登录验证

# 方案1：直接在业务逻辑里面修改，添加验证操作
# 缺点：复用性差、冗余度大、难以维护
```

025. Python函数-装饰器-案例-3

```python
# 定义两个功能函数
def send_shuoshuo():
    print('发说说')

def send_pic():
    print('发图片')

# 相关的逻辑代码
btn_index = 1
if btn_index == 1:
    send_shuoshuo()
elif btn_index == 2:
    send_pic()

# 发说说，发图片的前提：必须要登录验证

# 方案1：直接在业务逻辑里面修改，添加验证操作
# 缺点：复用性差、冗余度大、难以维护

# 方案2：直接在功能函数里面修改
# 优点：方便代码重用
# 缺点：功能函数多的话，仍然有冗余，复用性差，难以维护
```

```
# 方案3：单独写一个验证函数，在功能函数中调用
# 优点：可维护性高，代码重用
```

026. Python函数-装饰器-案例-4

赋值肯定是先计算右边的，右边计算好再给左边的。

```
# 定义两个功能函数
def send_shuoshuo():
    print('发说说')

def send_pic():
    print('发图片')

# 相关的逻辑代码
btn_index = 1
if btn_index == 1:
    send_shuoshuo()
elif btn_index == 2:
    send_pic()

# 发说说，发图片的前提：必须要登录验证

# 方案1：直接在业务逻辑里面修改，添加验证操作
# 缺点：复用性差、冗余度大、难以维护

# 方案2：直接在功能函数里面修改
# 优点：方便代码重用
# 缺点：功能函数多的话，仍然有冗余，复用性差，难以维护

# 方案3：单独写一个验证函数，在功能函数中调用
# 优点：可维护性高，代码重用
# 缺点：违背了开发封闭原则，也违背了单一职责原则

# 方案4：单独写一个验证函数，把功能函数传递进来
# 缺点：业务逻辑发生改变

# 方案5：使用闭包
print('-' * 30)

def check_login(func):
    def inner():
        print('登录验证成功！')
        func()
    return inner
```

```
send_shuoshuo = check_login(send_shuoshuo)
send_pic = check_login(send_pic)

btn_index = 1
if btn_index == 1:
    send_shuoshuo()
elif btn_index == 2:
    send_pic()

"""
------------------------------
登录验证成功!
发说说
"""

# 方案6: 语法糖写法
print('-' * 30)

def check_login(func):
    def inner():
        print('登录验证成功! ')
        func()
    return inner

@check_login
def send_shuoshuo():
    print('发说说')
# 等价于
# send_shuoshuo = check_login(send_shuoshuo)

@check_login
def send_pic():
    print('发图片')
# 等价于
# send_pic = check_login(send_pic)

btn_index = 1
if btn_index == 1:
    send_shuoshuo()
elif btn_index == 2:
    send_pic()
"""
------------------------------
登录验证成功!
发说说
"""
```

027. Python函数-装饰器-案例-5

```
def check(func):
    print('装饰器 check 执行')
    def inner():
        print('登录验证操作...')
        func()
    return inner

@check
def send_message():
    print('发说说')

# 给发说说增加一些额外的功能:
# 1. 函数名字不能发生改变
# 2. 函数体内部的代码不能发生改变

# send_message = check(send_message)
send_message()
"""
装饰器 check 执行
登录验证操作...
发说说
"""

# 当我们写 @check 的时候，等同于
# send_message = check(send_message)
# 所以装饰器 check 已经被立即执行
```

028. Python函数-装饰器-注意事项-1

```
# 装饰器的叠加

def decorator_line(func):
    def inner():
        print('-' * 30)
        func()
    return inner

def decorator_star(func):
    def inner():
        print('*' * 30)
        func()
    return inner
```

```
@decorator_line
@decorator_star
def print_content():
    print('大家好，我是警察')

print_content()
"""
------------------------------
******************************
大家好，我是警察
"""
```

029. Python函数-装饰器-注意事项-装饰器的执行图解

```
def dec(func):
    def inner():
        print('-' * 30)
        func()
    return inner

@dec
def pnum():
    print(10)

pnum()
"""
------------------------------
10
"""
```

030. Python函数-装饰器-注意事项-2

装饰带有参数的函数

```
def dec(func):
    def inner(*args, **kwargs):
        print('-' * 30)
```

```
            func(*args, **kwargs)
        return inner

    @dec
    def pnum(num):
        print(num)

    pnum(10)

    @dec
    def pnum2(num1, num2):
        print(num1, num2)

    pnum2(1, 4)
    """
    ------------------------------
    10
    ------------------------------
    1 4
    """
```

031. Python函数-装饰器-注意事项-3

装饰带有返回值的函数。
注意：

- `inner` 样式要与被装饰函数保持一致
- 被装饰函数有参数 – `inner` 要带参数
- 被装饰函数有返回值 – `inner` 要有返回值

这种相当于通用写法

```
def dec(func):
    def inner(*args, **kwargs):
        print('-' * 30)
        result = func(*args, **kwargs)
        return result
    return inner

@dec
def add(num1, num2):
    return num1 + num2

print(add(1, 2))
"""
------------------------------
```

```
3
"""

# 注意:
# inner 样式要与被装饰函数保持一致
# 被装饰函数有参数 -- inner 要带参数
# 被装饰函数有返回值 -- inner 要有返回值
```

---

032. Python函数-装饰器-注意事项-4

带有参数的装饰器。

> jpch89：带参数的装饰器，相当于装饰一个装饰器。
> 所以要多写一层。

```
# 带参装饰器

def dec_line(func):
    def inner():
        print('-' * 30)
        func()
    return inner

def dec_equal(func):
    def inner():
        print('=' * 30)
        func()
    return inner

def dec_start(func):
    def inner():
        print('*' * 30)
        func()
    return inner

@dec_line
def func():
    print('666')

func()

# 分析:
# 每个装饰器代码结构差不多
# 可以给装饰器传入一个字符参数
# 让它根据字符参数打印不同的分隔线
```

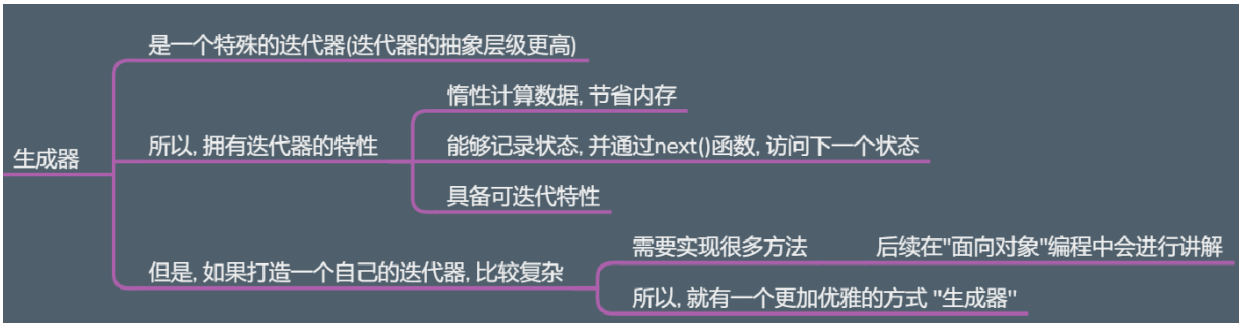
```python
# 给一个参数 char
# 返回一个新的装饰器

print('-' * 30)

def get_dec(char):
    def dec_char(func):
        def inner():
            print(char * 10)
            func()
        return inner
    return dec_char

@get_dec('呵')
def func():
    print('说点什么')

func()

"""
------------------------------
呵呵呵呵呵呵呵呵呵呵
说点什么
"""
```

总原则：把 `@` 后面的所有东西当做一个函数，即装饰器，来装饰后面紧跟着的函数。
即用 `get_dec('呵')` 来装饰 `func`。

> 个人理解：
>
> - 如果是 `get_dec` 不带参数，那么就是拿 `get_dec` 装饰 `func`
> - 如果是 `get_dec(参数)` 带参数，那么就是拿 `get_dec(参数)` 这个函数调用的执行结果（即返回值），来装饰 `func`
> - 根据参数的不同，生成了不同的装饰器，用来装饰 `func`

---

033. Python函数-生成器

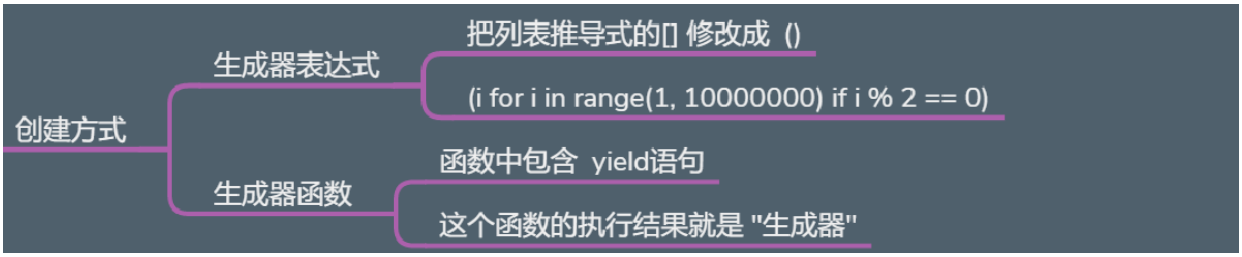


可以使用 `for ... in ...` 语法遍历的对象就是可迭代对象。

迭代器不一定是生成器，生成器一定是迭代器。
动物不一定是人，但是人一定是动物。
动物的抽象层级比人高。
同样，迭代器的抽象层级比生成器高。

---

034. Python函数-生成器-创建方式-1



通过下面这样的方式获取迭代器实际上有点问题，因为根据列表获取的迭代器，列表已经存在了，已经消耗了内存。

```
# 这样获取迭代器并没有减少内存消耗
l = [i for i in range(10000000)]
it = iter(l)
```

生成器的创建方式 `1` ：

```
g = (i for i in range(10000000) if i % 2 == 0)
print(g)
"""
<generator object <genexpr> at 0x0000019BF9EFA7D8>
"""

print(next(g))
print(next(g))
"""
0
2
"""

print(g.__next__())
"""
4
"""
```

---

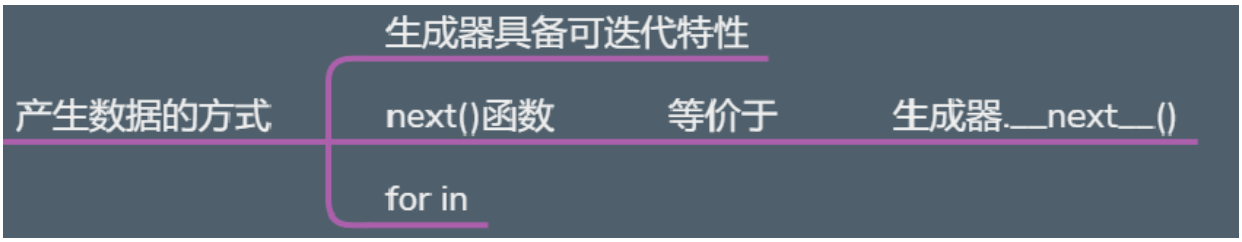
035. Python函数-生成器-创建方式-2

带有 `yield` 语句的函数叫做生成器函数
调用生成器函数，返回一个生成器
`yield` 后面就是状态值

`yield` 可以阻断当期函数执行，把后面的状态值返回给外界，之后对生成器使用 `next()` 函数时，或者调用生成器的 `__next__()` 方法时，函数会继续执行。
执行到下一个 `yield` 语句的时候，又会被暂停。

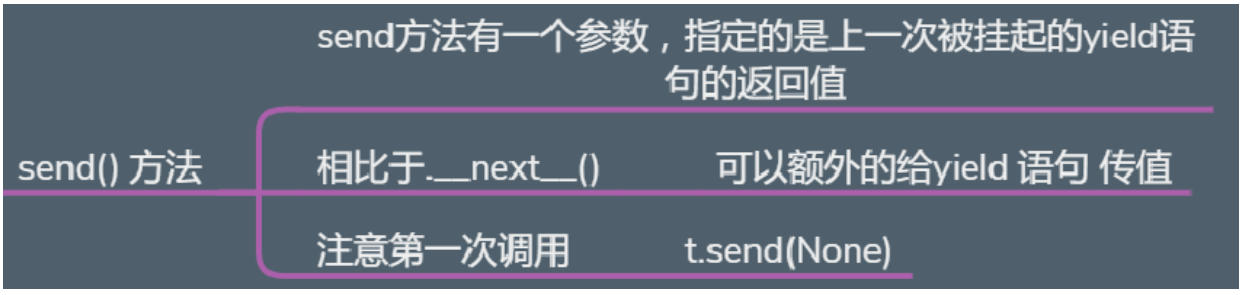
```
# 生成器的创建方式
def test():
    print('生成器开始')
    yield 1
    print('a')

    yield 2
    print('b')


g = test()
print(g)
"""
<generator object test at 0x0000022D30B9A7D8>
"""

# 带有 yield 语句的函数叫做生成器函数
# 调用生成器函数，返回一个生成器

print(next(g))
"""
生成器开始
1
"""
print(next(g))
"""
a
2
"""
print(next(g))
"""
b
Traceback (most recent call last):
  File "lk_030_生成器3.py", line 30, in <module>
    print(next(g))
StopIteration
"""
```

036. Python函数-生成器-访问方式




```
def test():
    for i in range(1, 9):
        yield i


g = test()
print(g)
"""
<generator object test at 0x000002027FDDA7D8>
"""

print(next(g))
"""
1
"""
print(g.__next__())
"""
2
"""
```

037. Python函数-生成器-send方法



> `next` 和 `send` 都可以推动生成器前进。

```
def test():
    print('生成器启动')
```

```
    res1 = yield 1
    print(res1)

    res2 = yield 2
    print(res2)

    res3 = yield 3
    print(res3)


g = test()
print(g.__next__())
"""
生成器启动
1
"""

print(next(g))
"""
None
2
"""

print(g.send('略略略'))
"""
略略略
3
"""

g = test()
# g.send('会报错')
"""
Traceback (most recent call last):
  File "lk_032_send.py", line 33, in <module>
    g.send('会报错')
TypeError: can't send non-None value to a just-started generator
"""

g.send(None)  # 等价于 g.next()
"""
生成器启动
"""
```

038. Python函数-生成器-close方法

关闭生成器

g.close()

后续如果继续调用, 会抛出StopIteration异常提示

```
def test():
    yield 1
    print('a')

    yield 2
    print('b')

    yield 3
    print('c')


g = test()

# 可以使用 next 函数、send 函数
# 可以调用 __next__ 方法、可以用 for in 遍历
print(g.__next__())
print(g.__next__())
print(g.__next__())
"""
1
a
2
b
3
"""

# print(g.__next__())  # 会报错
"""
c
Traceback (most recent call last):
  File "lk_033_close.py", line 27, in <module>
    print(g.__next__())  # 会报错
StopIteration
"""

# 使用 close 提前关闭生成器
g = test()
print(next(g))
"""
1
"""
g.close()
print(next(g))
"""
Traceback (most recent call last):
  File "lk_033_close.py", line 43, in <module>
```

```
    print(next(g))
StopIteration
"""
```

039. Python函数-生成器-注意事项

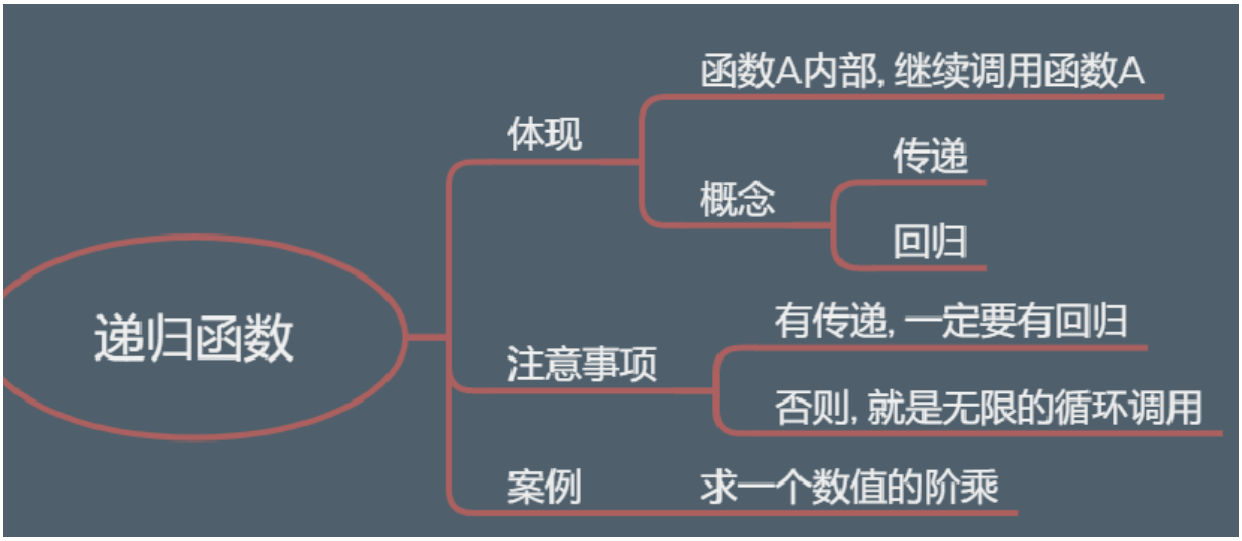
碰到 `return` 会立即抛出 `StopIteration` 异常，并把 `return` 后面的东西作为异常的提示信息显示出来。

```
def test():
    yield 1
    print('a')
    yield 2
    print('b')
    return '一则消息'


g = test()
print(next(g))
"""
1
"""

print(next(g))
"""
a
2
"""

print(next(g))
"""
b
Traceback (most recent call last):
  File "lk_034_return.py", line 21, in <module>
    print(next(g))
StopIteration: 一则消息
"""
```

生成器只会被遍历一次。

```python
def test():
    yield 1
    print('a')
    yield 2
    print('b')
    yield 3
    print('c')


g = test()
for i in g:
    print(i)
print('-' * 20)
for i in g:
    print(i)
"""
1
a
2
b
3
c
--------------------
"""
```

040. Python函数-递归函数



四个小朋友。
第一个小朋友两个糖。
告诉第二个小朋友：你的糖是前面小朋友的两倍。
同样告诉第三个、第四个小朋友。
此时直接问第四个小朋友，你会得到几颗糖啊？

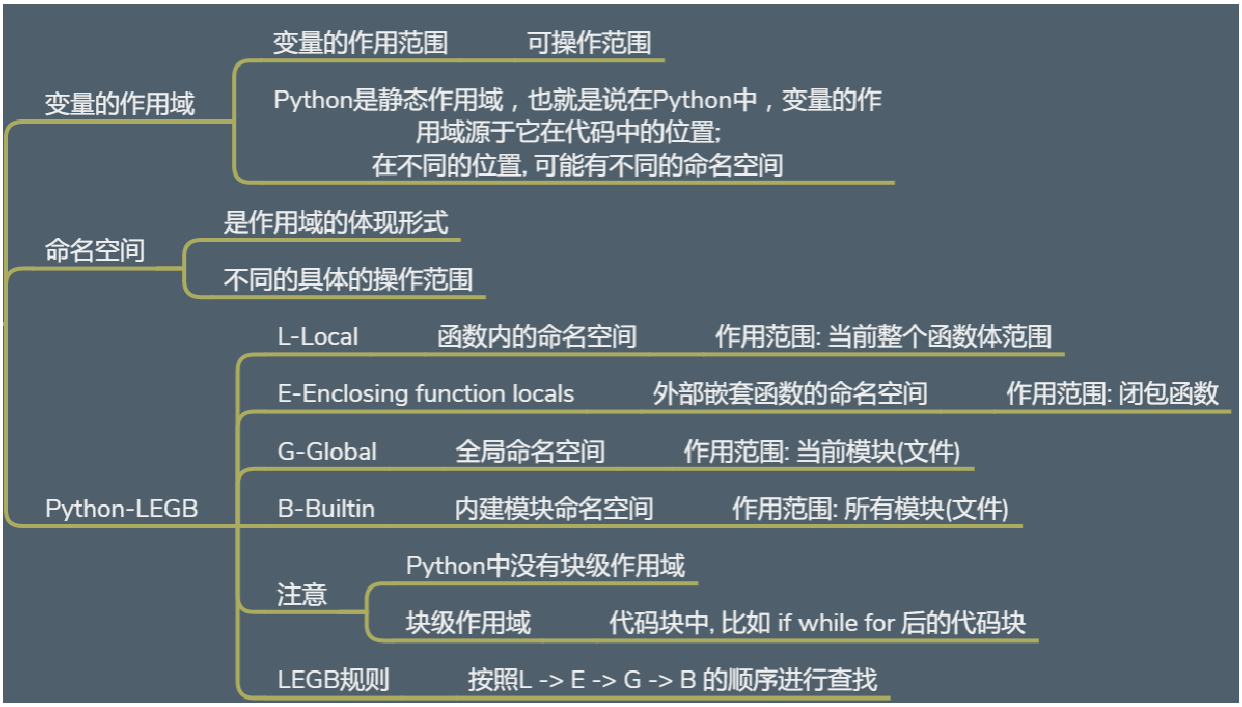
传递（把问题传递下去）
第四个小朋友要去问第三个小朋友，他有几颗糖。
同样，第三个问第二个。
第二个问第一个。

回归（让结果回归回来）
第一个告诉第二个小朋友，我有两颗糖。
第二个告诉第三个，我有四颗糖。
第三个告诉第四个，我有八颗糖。
第四个说出答案，我有十六颗糖。

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

res = factorial(5)
print(res)
```

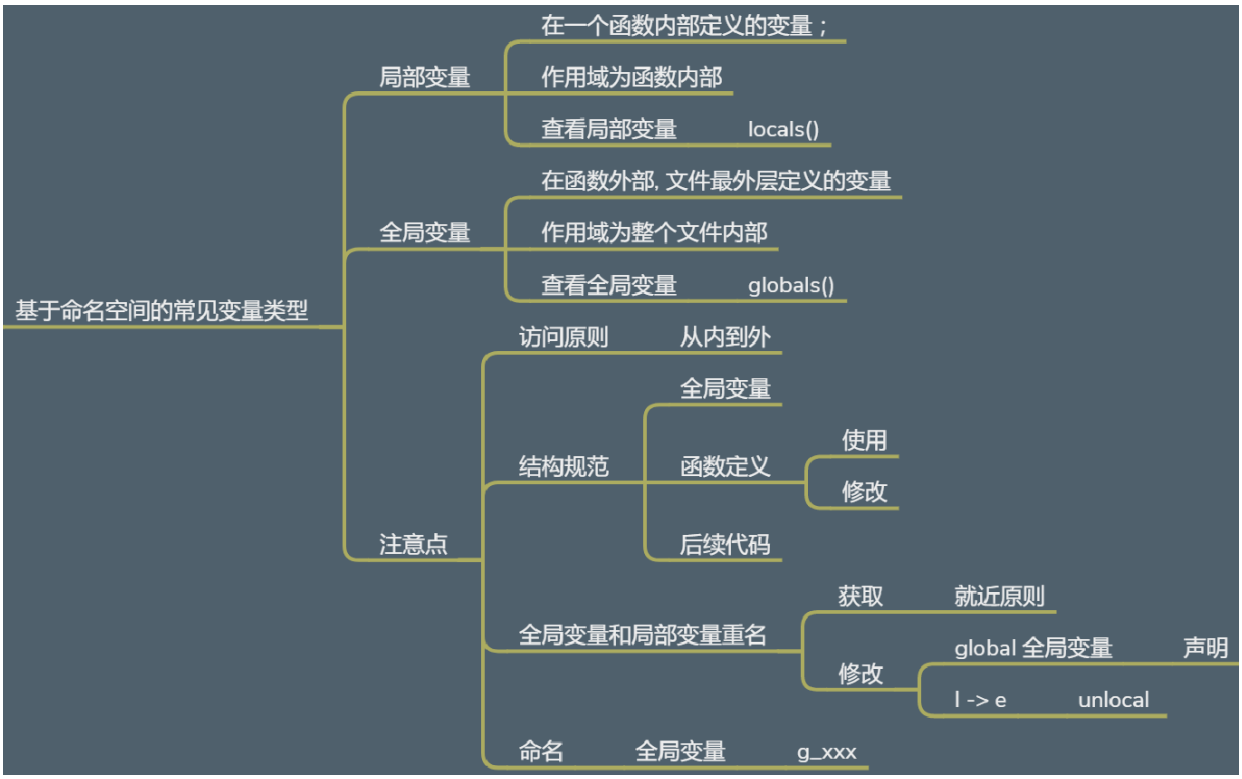
---

041. Python函数-作用域-概念



---

042. Python函数-作用域-局部变量-全局变量



> 【注】
> 图片右下方不应该是 `unlocal` 而应该是 `nonlocal`

`locals()` 查看局部变量，返回一个字典
`globals()` 查看全局变量。

全局变量要写在文件的最上面！

```python
# 迟绑定
a = 1

def test():
    print(a)
    print(b)

b = 2
test()
"""
1
2
"""
# 个人理解:
# 迟绑定就是调用时取值，而不是定义时取值。
```

但是这样写不行：

```
c = 1

def test():
    print(c)
    print(d)

test()
d = 2
"""
1
Traceback (most recent call last):
  File "lk_036_变量.py", line 31, in <module>
    test()
  File "lk_036_变量.py", line 29, in test
    print(d)
NameError: name 'd' is not defined
"""
```

调用 `test` 的时候取值，想要取 `d` 的值，但是没有取到。
因为代码还没有运行到 `d = 2` 这一行。
为了避免这个问题，可以**把所有的全局变量写在最上面。**

完成于 `2018.11.27 0:00`