Python

Function

Parameters 形参

普通参数, `args` . `**kwargs`

- 普通参数：就是确定的参数
- `*arg`：不定长参数，未知参数名
- `**kwargs`：不定长参数，已知参数名，就是字典（不能和普通参数重名）

```
def test(name, value, *arg, **kwargs):
    print(name, value)
    print(arg)
    print(kwargs)

test("hello", "123", 1,2,3,4, verbose=23)
# =>
# hello 123
```

```
# (1, 2, 3, 4)
# {'verbose': 23}
```

💡 参数种类顺序不能错，必须是 普通参数 ⇒ `*arg` ⇒ `**kwargs`

Closure 闭包

使用用途：

1. 读取函数内部的变量
2. 防止局部变量被回收

💡 就是把函数当作变量使用，函数式编程

```
# 防止被回收
def create(pos=[0,0]):

    def go(direction, step):
        new_x = pos[0]+direction[0]*step
        new_y = pos[1]+direction[1]*step

        pos[0] = new_x
        pos[1] = new_y

        return pos


    return go

player = create()
print(player([1,0],10))
print(player([0,1],20))
print(player([-1,0],10))

# --------------------------------

# 可以无数嵌套
def w1(k1=1):
    def w2(k2=2):
        def w3(k3=10):
            return k1 * k2 * k3
```

```
        return w3
    return w2

ret = w1(2)(3)(5)  # => 30
```

💡 实际上功能和类差不多

Decorator 装饰器

可能是闭包最常用的用途

Implemented By Function

```
# decorator without parameters
def timer1(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print(end - start)
    return wrapper

@timer1
def hi(name):
    print(name)

hi("hi")
# =>
# hi
# 1.6689300537109375e-06
```

```
# decorator with parameters
def timer2(base=1):
    def wrapper1(func):
        def wrapper2(*args, **kwargs):
            start = time.time()
            func(*args, **kwargs)
            end = time.time()
            print(end - start + base)
```

```
        return wrapper2

    return wrapper1


@timer2(base=2)
def hi(name):
    print(name)


hi("test")
# =>
# test
# 2.0000014305114746
```

💡 最多三层函数，第二层输入被装饰函数

Implemented By Class

```
class Timer(object):
    def __init__(self, k=1):
        self.k = k
        pass

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            func(*args, **kwargs)
            self.hello()

        return wrapper

    def hello(self):
        print("this is a decorator class %d" % self.k)

# pay attention to the parentheses
@Timer()
def timer_test(name):
    print(name)


timer_test("timer")
# =>
# timer
# this is a decorator class 1
```

Wraps

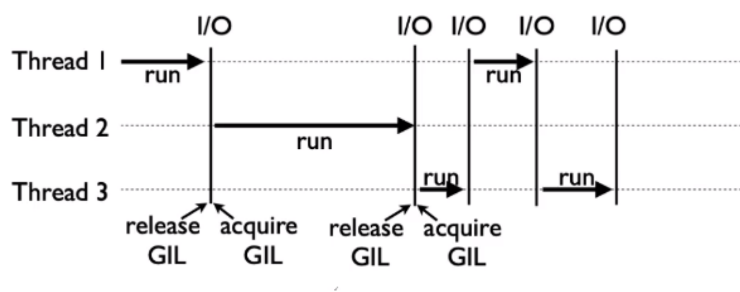
装饰之后函数的 `__name__` , `__doc__` 等属性会跟随装饰器函数，若想保留原函数属性则用这个

```
from functools import wraps

# without wraps
def timer1(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print(end - start)
    return wrapper

@timer1
def hi(name):
    print(name)

print(hi.__name__) #  => wrapper

# ---------------------------------

# with wraps
def timer1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print(end - start)
    return wrapper

@timer1
def hi(name):
    print(name)

print(hi.__name__) #  => hi
```

Concurrency & Parallelism & Coroutine

- `threading` ： 多线程，但实际上python**不存在真正的多线程**
    - 优：占用资源比进程少

- 劣：占用资源比协程多（线程上下文切换）
- 适用于 IO-bound



- `multiprocessing` ：多进程，可以并行运算
  - 优：可以利用多核实现并行（但不是开几个线程就是用几核）
  - 列：开销大
  - 适用于 CPU-bound
- `asyncio` ：协程，单线程中异步执行，功能类似多线程
  - 优：开销最小，可以启动大量协程
  - 列：第三方库必须支持协程才能使用，代码较复杂
  - 适用于 IO-bound

> 对于多重循环的计算密集型任务，多线程还是可能比单线程快

Concurrency & Parallelism

`threading` 和 `multiprocessing` 用法基本一样

语法条目	多线程	多进程
引入模块	from threading import Thread	from multiprocessing import Process
新建 启动 等待结束	t=Thread(target=func, args=(100,)) t.start() t.join()	p = Process(target=f, args=('bob',)) p.start() p.join()
数据通信	import queue q = queue.Queue() q.put(item) item = q.get()	from multiprocessing import Queue q = Queue() q.put([42, None, 'hello']) item = q.get()

线程安全加锁	from threading import Lock
	lock = Lock()
	with lock:
	    # do something
		from multiprocessing import Lock
		lock = Lock()
		with lock:
		    # do something

池化技术	from concurrent.futures import ThreadPoolExecutor
	
	with ThreadPoolExecutor() as executor:
	    # 方法1
	    results = executor.map(func, [1,2,3])
	
	    # 方法2
	    future = executor.submit(func, 1)
	    result = future.result()
		from concurrent.futures import ProcessPoolExecutor
		
		with ProcessPoolExecutor() as executor:
		    # 方法1
		    results = executor.map(func, [1,2,3])
		
		    # 方法2
		    future = executor.submit(func, 1)
		    result = future.result()

Future

多线程或多进程里面某个线程或进程的句柄，可以用这个句柄查看该线程或进程的状态、属性等

`submit` 返回的是 `Future`

`Future.result()`：阻塞地获取该Future的结果

池化技术

> 💡 似乎 `Executor` 的开销十分大， `ProcessPoolExecutor` 速度比不过单线程，然而 `multiprocessing.Pool` 可以（不知道什么原因）

Executor

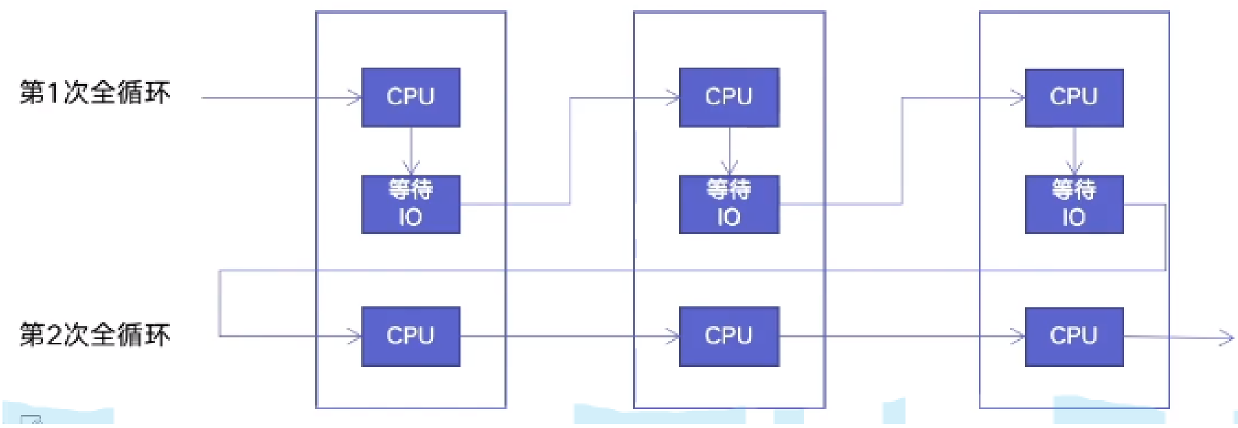
```
pool = ProcessPoolExecutor(4)
rets = pool.map(is_prime, numbers)  # map is blocking, return a generator
pool.shutdown()  # block
```

Pool

`threading` 里没有 `Pool` ，只有 `multiprocessing` 有

```
pool = mp.Pool(4)
rets = pool.map(is_prime, numbers)  # map is blocking, return a generator
pool.close()
pool.join()
```

Coroutine



用一个大循环重复执行单线程内的不同任务，当某个任务需要等待，则跳过这个任务去执行另一个

> 💡 有点像 js 里面异步代码的执行顺序

```
import asyncio

async def async_method(begin, end):
    for i in range(begin, end):
        tasks[i] **= 2
    await asyncio.sleep(3)  # 这里写被阻塞的东西

s = time.time()

loop = asyncio.get_event_loop()
ts = [loop.create_task(async_method(i * l, (i + 1) * l)) for i in range(3)]
loop.run_until_complete(asyncio.wait(ts))
```

```
e = time.time()
print(e - s)
```

PriorityQueue

```
# 1. prior: small prior's value means it is on the top of queue
q = PriorityQueue()

# method 1: use tuple
# object in PriorityQueue is a tuple and the first item is prior, the second is your value
q.put((1, "a"))
q.put((3, "b"))
q.get() # => "a"

# method 2: custom class with __lt__ overrided
class Node:
    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        return self.value < other.value

q.put(Node(10))
q.put(Node(9))
q.get() # => Node(9)
```