变量赋值

- 可变数据类型：
  - 列表 list
  - 字典 dict

- 不可变数据类型：
  - 整型 int
  - 浮点型 float
  - 字符串型 string
  - 元组 tuple

作为函数参数传递的区别

- 关键代码

```
def __init__(self, name, goods=None):
    self.name = name
    if goods is None:
        goods = []
    self.goods = goods
```

序列

序列分类

- 容器序列：list、tuple、collections.deque 等，能存放不同类型的数据 容器序列可以存放不同类型的数据。
- 扁平序列：str、bytes、bytearray、memoryview (内存视图)、array.array 等，存放的是相同类型的数据 扁平序列只能容纳一种类型。

容器序列存在深拷贝、浅拷贝问题

- 注意：非容器（数字、字符串、元组）类型没有拷贝问题

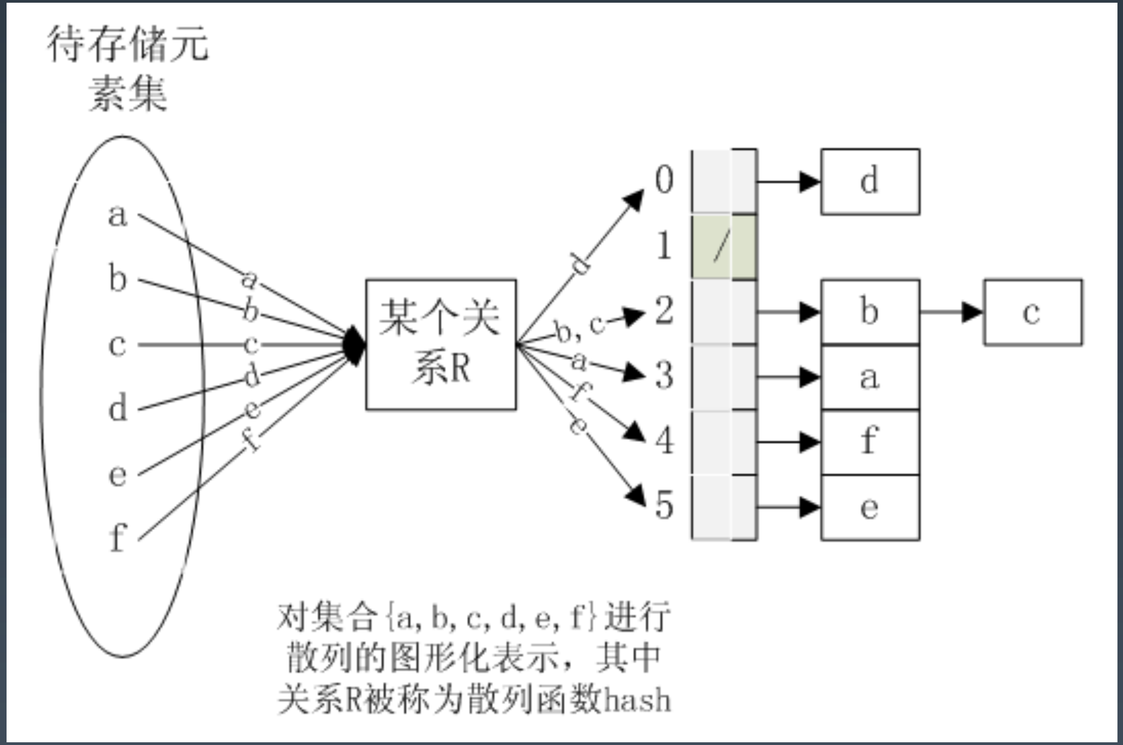
```
import copy
copy.copy(object)
copy.deepcopy(object)
```

序列

另一种分类方式

- 可变序列 list、bytearray、array.array、collections.deque 和 memoryview。
- 不可变序列 tuple、str 和 bytes。

字典与哈希


待存储元
素集
a
b
c
d
e
f
某个关
系R
0
1
2
3
4
5
d
b, c
a
f
e
d
b
c
a
f
e
对集合{a, b, c, d, e, f}进行
散列的图形化表示，其中
关系R被称为散列函数hash

使用 collections 扩展内置数据类型

collections 提供了加强版的数据类型

https://docs.python.org/zh-cn/3.6/library/collections.html

namedtuple -- 带命名的元组

```
import collections
Point = collections.namedtuple('Point', ['x', 'y’])
p = Point(11, y=22)
print(p[0]  + p[1])
x, y = p
print(p.x + p.y)
print(p)
```

deque 双向队列

Counter 计数器

函数

函数需要关注什么

- 调用
- 作用域
- 参数
- 返回值

变量作用域

高级语言对变量的使用：

- 变量声明
- 定义类型（分配内存空间大小）
- 初始化（赋值、填充内存）
- 引用（通过对象名称调用对象内存数据）

Python 和高级语言有很大差别，在模块、类、函数中定义，才有作用域的概念。

变量作用域

Python 作用域遵循 LEGB 规则。

LEGB 含义解释：

- L-Local(function)；函数内的名字空间
- E-Enclosing function locals；外部嵌套函数的名字空间（例如closure）
- G-Global(module)；函数定义所在模块（文件）的名字空间
- B-Builtin(Python)；Python 内置模块的名字空间

变量赋值

面试题 1

```
a = 123
b = 123
c = a
a = 456
c = 789
c = b = a
```

基本数据类型的可变类型与不可变类型区分

面试题 2

```
a = [1,2,3]
b = a
a.append(4)
print(b)
```

基本数据类型的可变类型与不可变类型区分

面试题 3

```
a = [1, 2, 3]
b = a
a = [4, 5, 6]
a 和 b 分别是什么值?

a = [1, 2, 3]
b = a
a[0],a[1],a[2] = 4, 5, 6
a 和 b 分别是什么值?
```

面试题 4

思考哪种类型可以做字典的 key？为什么？

参数

- 参数分类
  - 必选参数
  - 默认参数
  - 可变参数
  - 关键字参数
  - 命名关键字参数

- 参数是函数-高阶函数
- 有些函数比较简单，封装成 Lambda 表达式

函数的可变长参数

一般可变长参数定义如下：

```
def func(*args, **kargs):
    pass
```

kargs　获取关键字参数
args　获取其他参数

示例：

```
def func(*args, **kargs):
    print(f'args: {args}’)
    print(f'kargs:{kargs}’)

func(123, 'xyz', name='xvalue' )
```

Lambda 表达式

Lambda 只是表达式，不是所有的函数逻辑都能封装进去

```
k = lambda x:x+1
print(k(1))
```

Lambda 表达式后面只能有一个表达式

- 实现简单函数的时候可以使用 Lambda 表达式替代
- 使用高阶函数的时候一般使用 Lambda 表达式

偏函数

functools.partial：返回一个可调用的 partial 对象

使用方法：partial(func,*args,**kw)

注意：

- func 是必须参数
- 至少需要一个 args 或 kw 参数

返回值

- 返回的关键字
  - return
  - yield

- 返回的对象
  - 可调用对象--闭包(装饰器)

高阶函数

高阶：参数是函数、返回值是函数

常见的高阶函数：map、reduce、filter、apply

apply 在 Python2.3 被移除，reduce 被放在 functools 包中

推导式和生成器表达式可以替代 map 和 filter 函数

高阶函数

map (函数， 序列) 将序列中每个值传入函数，处理完成返回为 map 对象

```
number = list(range(11))

def square(x):
    return x**2

print(list(map(square, number)))

print(dir(map(square, number)))
```

filter (函数，序列)将序列中每个值传入函数，符合函数条件的返回为 filter 对象

装饰器

增强而不改变原有函数

装饰器强调函数的定义态而不是运行态

装饰器语法糖的展开：

```
@decorate
def target():
    print('do something')

def target():
    print('do something')
target = decorate(target)
```

装饰器

target 表示函数

target() 表示函数执行

new = func 体现“一切皆对象”，函数也可以被当做对象进行赋值

被装饰函数

被修饰函数带参数

被修饰函数带不定长参数

被修饰函数带返回值

装饰器

装饰器带参数

装饰器堆叠

内置的装饰方法函数

functools.wraps

functools.lru_cache

装饰器对类的支持

类装饰器（类的函数装饰器）

装饰类

DataClass Python3.7 支持的新装饰器

其他内置类装饰器

- classmethod
- staticmethod
- property

对象协议

Duck Typing 的概念

容器类型协议

- __str__ 打印对象时，默认输出该方法的返回值
- __getitem__、__setitem__、__delitem__ 字典索引操作
- __iter__ 迭代器
- __call__ 可调用对象协议

对象协议

比较大小的协议

- __eq__
- __gt__

描述符协议和属性交互协议

- __get__
- __set__

可哈希对象

- __hash__

上下文管理器

with 上下文表达式的用法

使用 __enter__() __exit__() 实现上下文管理器

协程

- 协程和线程的区别是什么
- yield/send 与 yield from 作为协程如何使用
- asyncio 模块
- asyncio.coroutine 和 yield from 的关系
- 事件循环机制

协程和线程的区别

- 协程是异步的，线程是同步的
- 协程是非抢占式的，线程是抢占式的
- 线程是被动调度的，协程是主动调度的
- 协程可以暂停函数的执行，保留上一次调用时的状态，是增强型生成器
- 协程是用户级的任务调度，线程是内核级的任务调度
- 协程适用于 IO 密集型程序，不适用于 CPU 密集型程序的处理

迭代器

- 迭代器与可迭代的区别
- 利用 yield 实现生成器
- itertools 库的用法

生成器-1

1. 在函数中使用 yield 关键字，可以实现生成器。
2. 生成器可以让函数返回可迭代对象。
3. yield 和 return 不同，return 返回后，函数状态终止，yield 保持函数的执行状态，返回后，函数回到之前保存的状态继续执行。
4. 函数被 yield 会暂停，局部变量也会被保存。
5. 迭代器终止时，会抛出 StopIteration 异常。

生成器-2

```
print([ i for i in range(0,11)])
```

替换为

```
print(( i for i in range(0,11)))
```

```
gennumber = ( i for i in range(0,11))
print(next(gennumber))
print(next(gennumber))
print(next(gennumber))
# print(list(gennumber))
print([i for i in gennumber ])
```

生成器-3

Iterables： 包含 __getitem__() 或 __iter__() 方法的容器对象

Iterator： 包含 next() 和 __iter__() 方法

Generator： 包含 yield 语句的函数

yield from

yield from 是表达式，对 yield 进行了扩展

```
def ex1():
    yield 1
    yield 2
    return 3

def ex2():
    ex1_result = yield from ex1()
    print(f'ex1 : {ex1_result}')
    yield None
```

异步编程

python3.5 版本引入了 await 取代 yield from 方式

```
import asyncio
async def py35_coro():
    await stuff()
```

注意： await 接收的对象必须是 awaitable 对象

awaitable 对象定义了 __await__() 方法

awaitable 对象有三类：

1. 协程 coroutine
2. 任务 Task
3. 未来对象 Future

aiohttp

aiohttp 异步的 HTTP 客户端和服务端

```
from aiohttp import web

# views
async def index(request):
    return web.Response(text='hello aiohttp')

# routes
def setup_routes(app):
    app.router.add_get('/', index)

# app
app = web.Application()
setup_routes(app)
web.run_app(app, host='127.0.0.1', port=8080)
```

THANKS!