Python 入门与进阶技巧

软件03 和嘉晅 jx-he19@mails.tsinghua.edu.cn

内容速览

- 基础语法与数据结构
- 鸭子类型（Duck Typing）
- 模块
- 装饰器
- 并发编程
- 杂项

基础语法与数据结构

把Python当计算器用

- 安装后在命令行输入 python ，启动一个交互式Python解释器。
- 支持加减乘除，整除，模数，乘法等运算
- 创建 list, tuple, dict

基础数据结构

- int，float，str，bool，NoneType
- list，tuple：可变与不可变列表
- dict：字典

控制流语句

基础：if，elif，else，while，for

```
words = ['cat', 'window', 'dog']
for w in words:
    print(w, len(w))
for i in range(0, len(words))
    print(i, words[i])
for (i, w) in enumerate(words):
    print(i, w)
```

match语句

match-case类似于C中的switch-case语句，但是match-case还有模式匹配的功能，用起来更加灵活。

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

函数定义与函数调用

通过 def 定义函数，在括号中写参数调用。

```
def fib(n):
    if n == 1 or n == 0:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(10))
```

函数的更多打开方式

默认值参数

```
def fib(n = 10):
    if n == 1 or n == 0:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib())
```

关键字参数

```
def print_str(a, b='Hello World!', c='QwQ'):
    print(a, b, c)
print_str('1')
print_str('1', '2')
print_str('1', c='2')
# print_str(c='2', '1') SyntaxError: positional argu
```

函数的更多打开方式

***args**

```
def average(*args):
    ret = 0
    for arg in args:
        ret += arg
    return ret / len(args)

print(average(1,2,3))
print(average(*[2,3,4]))
```

****kwargs**

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))

myFun(first='1', mid='2', last='3')
myFun(**{'first': '1', 'mid': '2', 'last': '3'})
```

lambda表达式

```
def make_incrementor(n):
     return lambda x: x + n

f = make_incrementor(42)
print(f(0))
print(f(1))
```

```
data = [
    {
        'name': 'Alice',
        'scores': [100, 97, 98],
    },
    {
        'name': 'Bob',
        'scores': [99, 99, 94]
    }
]
data.sort(key=lambda data: sum(data['scores']))
print(data)
```

类定义与实例化

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
    def print(self):
        print(f'{self.r} + {self.i}i')

c = Complex(2, 3)
c.print()  # 2 + 3i
```

- `class ClassName(BaseClass)` 可以继承（或多重继承）
- Python类中没有私有变量与成员函数，约定在名称前加下划线表示私有。

更多相关内容请参考Python官方教程

鸭子类型（Duck Typing）

什么是鸭子类型（Duck Typing）

如果有一个东西，走起来像鸭子、游泳像鸭子、叫起来也像鸭子，那它就是一只鸭子。

```
class Duck:
    def swim(self):
        print("Duck swimming")

    def fly(self):
        print("Duck flying")

class Whale:
    def swim(self):
        print("Whale swimming")

duck_list = [Duck(), Whale()]
for duck in duck_list:
    duck.swim()
    # duck.fly(): AttributeError: 'Whale' object has no attribute 'fly'
```

C++中的抽象类

相比于Python可以用Duck Typing隐式地实现多态，C++需要显式地定义出接口。

```
#include <iostream>

class Animal {
    public:
        virtual void swim() = 0;
        virtual void fly() = 0;
};
```

```
class Duck: Animal {
    public:
        void swim() { std::cout << "Duck swimming" <
        void fly() { std::cout << "Duck flying" << s
};

class Whale: Animal {
    public:
        void swim() { std::cout << "Whale swimming"
}; // 未实现fly，不能实例化
```

实例：Sum函数

NumPy的文档中如此描述sum的第一个位置参数 a: array_like, Elements to sum.

NumPy.array中对 array_like 的解释如下： An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence. If object is a scalar, a 0-dimensional array containing object is returned.

```
import numpy as np
import pandas as pd

print(np.sum(np.ones((4, 4))))       # Output: 16.0
print(np.sum([1, 2, 3, 4]))          # Output: 10
print(np.sum(pd.Series([2,3,4,5])))  # Output: 14
```

Python中的抽象类

Duck不必。

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass

    @abstractmethod
    def write(self, data):
        pass

class SocketStream(IStream):
    def read(self, maxbytes=-1):
        pass

    def write(self, data):
        pass
```

模块

创建与引入单个模块

- 一个以 .py 结尾的文件就是一个Python模块
- 用 import ... 导入，导入的模块被加载（执行），以供调用

文件结构

```
a.py
b.py
```

a.py

```
import b  # bar

b.foo()  # foo
obj = b.B()  # Creating B
print(b.var)  # qwq
```

b.py

```
def foo():
    print("foo")

class B:
    def __init__(self):
        print("Creating B")

var = "qwq"

print("bar")
```

多级模块

- 在目录下创建一个空白的 __init__.py 就可以把一个目录声明为一个Python模块。
- 目录中包含的模块文件（.py 或有 __init__.py 的文件夹）成为目录模块的子模块，可以通过 import a.b 的形式导入。

文件结构

```
main.py
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

main.py

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

__init__.py

文件结构

```
main.py
torch/
    __init__.py
    nn.py
```

main.py

```
import torch  # 这里会导入同目录下的torch模块
model = torch.nn.Linear()
```

torch/nn.py

```
class Linear():
    def __init__(self):
        print('Linear Model.')
```

例：torch.nn中 __init__.py 的编写。

导入路径

- 导入一个模块时，解释器首先搜索内置模块（标准库）。如果没有找到，解释器就在变量 `sys.path` 给出的目录列表中搜索
- sys.path 包括：
  - 输入脚本的目录
  - 环境变量PATH：大部分第三方库都包含在这里
  - site-packages

相对导入

- 不同于绝对导入，相对导入通过相对于当前模块（ __name__ ）的相对路径导入
- __name__ 在被作为模块调用时为具体模块的名称（例："foo.bar"），作为脚本执行时为"__main__"

文件结构

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

mypackage/A/spam.py

```
from ..B import bar
# from mypackage.B import bar
```

注意：相对导入需要考虑 __name__ == '__main__'的特殊情况。

参考资料

- Python 语言参考手册 - 导入系统
- Python CookBook - 模块与包

装饰器

什么是装饰器

- 装饰器以 @ 符号开始，放在函数前，可以修饰一个函数。
- 许多Python库中都能看到装饰器的身影
  - Flask中用装饰器将一个视图函数挂载到路由上
  - Python内置 @staticmethod， @classmethod 声明类静态方法
  - Python内置 @property， getter, setter, deleter 声明类属性

```
from functools import cache

@cache
def fib(n):
    return fib(n - 1) + fib(n - 2) if n > 1 else n

print(fib(40)) # 不加cache 18s，加cache秒出结果
```

编写装饰器

- 编写装饰器：编写一个包装函数的函数
- funtools库提供了 wraps 装饰器来简化我们的任务

```
import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper

@timethis
def work():
    ... ...
```

带参数的装饰器

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y
```

并发编程

创建线程/进程

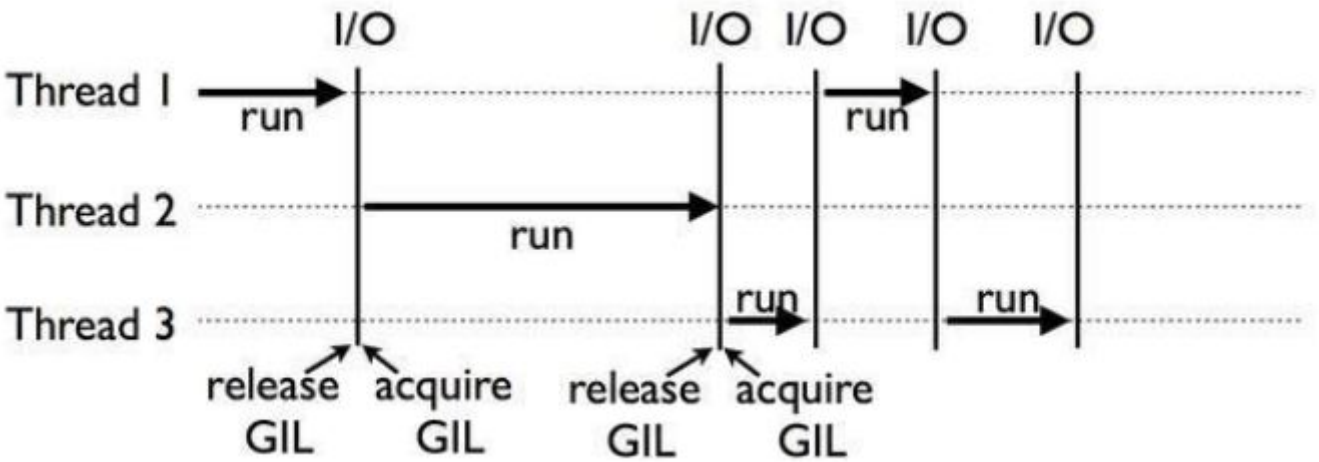
```
import time
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(1)

# thread
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
# process
from multiprocessing import Process
p = Process(target=countdown, args=(10, ))
p.start()
```

“优化”计算密集型任务

```
from sympy import isprime
from threading import Thread

THREAD_NUMBER = 10
results = [None for i in range(THREAD_NUMBER)]

def prime_count(begin, end, index):
    count = 0
    is_prime_list = [1 if isprime(number) else 0 for number in range(begin, end)]
    results[index] = sum(is_prime_list)

n = 1000000
t_pool = [
    Thread(target=prime_count, args=(1 + i * n // THREAD_NUMBER, 1 + (i + 1) * n // THREAD_NUMBER, i))
    for i in range(0, THREAD_NUMBER)
]
list(map(lambda t: t.start(), t_pool))
list(map(lambda t: t.join(), t_pool))
print(sum(results))
```

全局解释器锁（GIL）

一个CPython解释器进程中，任何时候都只有一个Python线程执行



用进程优化计算密集型任务

```
from sympy import isprime
from multiprocessing import Pool

PROCESS_NUMBER = 16
n = 1000000

with Pool(PROCESS_NUMBER) as pool:
    print(sum(pool.map(isprime, range(1, n + 1))))
```

第3个选择：协程

- 协程（coroutine），也被称为微线程，是一种用户态内的上下文切换技术
- 多个协程之间往往由单个线程，用事件循环来调度
- Python使用 async def 关键字定义协程函数，在协程函数中使用 await 关键字定义中断点

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")
    await asyncio.gather(say_after(1, 'hello'), say_after(2, 'world'))
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

为什么使用协程

- 轻量化，易于编写IO密集型代码
- 大量第三方库支持：awesome-asyncio
- 可替换事件循环：uvloop

杂项

- 环境管理：pip，虚拟环境，conda
- Python 官方教程与Python CookBook
- 去读第三方库的文档吧