量化金融 python 基础 6-Function

1 Function 函数

函数是用来执行特定任务的代码块，

特别是需要多次重复执行某项特定工作时，

调用封装好的函数，可以使程序更容易读写、测试和维护。

1.1 1. Introduction to Function

[1]:
```
# 定义一个函数，计算默认窗口期为 5 的移动平均数

def find_rolling_mean(prices: list, window_size: int = 5):
    rolling_means = []
    for i in range(len(prices) - window_size + 1):
        price_piece = prices[i:i+window_size]
        mean = sum(price_piece) / window_size
        rolling_means.append(mean)

    return rolling_means
```

[2]:
```
prices = [20, 3, 5, 8, 7, 52, 6, 45]
window_size = 5
```

[3]:
```
find_rolling_mean(prices, window_size)
```

[3]:
```
[8.6, 15.0, 15.6, 23.6]
```

1.2 2. 匿名函数 lambda

lambda 在基础 4 的课程中介绍过，
这里可以 lambda 简化函数

[4]:
```
def find_rolling_mean2(prices: list, window_size: int):
    rolling_means = []
    average = lambda price_piece: sum(price_piece) / len(price_piece)
    for i in range(len(prices) - window_size + 1):
        rolling_means.append(average(prices[i: i+window_size]))

    return rolling_means

find_rolling_mean2(prices, window_size)
```

[4]:
```
[8.6, 15.0, 15.6, 23.6]
```

[5]:
```
# simplyfy further
# function: find_rolling_mean3
[(lambda price_piece: sum(price_piece) / window_size)(prices[i:i+window_size])\
 for i in range(len(prices) - window_size + 1)]
# lambda 函数内输入一个 price_piece 变量
# 该变量在第二个小括号内，指代 prices[i:i+window_size]
# 在 lambda 函数内计算 prices[i::i+window_size] 的平均价格
```

[5]:
```
[8.6, 15.0, 15.6, 23.6]
```

1.3 3. 输入任意数量参数 Arbitrary number of arguments

一次性求多个窗口期的滚动平均

[6]:
```
def find_rolling_means(prices: list, *window_sizes):
    # window_sizes 是一个元组 tuple，可以存放多个窗口周期用于计算。
    rolling_means: dict[int, list] = {}
    for window_size in window_sizes:
        rolling_means[window_size] = []
        for i in range(len(prices) - window_size + 1):
            price_piece = prices[i:i+window_size]
            mean = sum(price_piece) / window_size
```

```
            rolling_means[window_size].append(mean)

    return rolling_means
```

[7]:
```
window_sizes = (3, 4, 5)

find_rolling_means(prices, *window_sizes)
```

[7]:
```
{3: [9.333333333333334,
  5.333333333333333,
  6.666666666666667,
  22.333333333333332,
  21.666666666666668,
  34.333333333333336],
 4: [9.0, 5.75, 18.0, 18.25, 27.5],
 5: [8.6, 15.0, 15.6, 23.6]}
```

[8]:
```
# 偷个懒可以直接调用第一个定义的函数，find_rolling_mean.
def find_rolling_means2(prices: list, *window_sizes):
    rolling_means: dict[int, list] = {}
    for window_size in window_sizes:
        rolling_means[window_size] = find_rolling_mean(prices, window_size)

    return rolling_means
```

[9]:
```
find_rolling_means2(prices, *window_sizes)
```

[9]:
```
{3: [9.333333333333334,
  5.333333333333333,
  6.666666666666667,
  22.333333333333332,
  21.666666666666668,
  34.333333333333336],
 4: [9.0, 5.75, 18.0, 18.25, 27.5],
 5: [8.6, 15.0, 15.6, 23.6]}
```

1.4 4. 任意数量的键参数 Arbitrary key arguments

[10]:
```python
import math

def find_indicators(prices: list, **kwargs):
# kwargs 全称是 keyword arguments,
# 它是一个字典，字典的 key 是变量名，字典的 value 存放参数。
    if "ma" in kwargs:
        window = kwargs["ma"]
        ma = sum(prices[-window:]) / window
        print(f"Moving Average ({window}) is {ma}")

    if "volatility" in kwargs:
        window = kwargs['volatility']
        mean = sum(prices) / len(prices)
        squared_diff_sum = sum((price - mean) ** 2 for price in prices)
        mean_squared_diff = squared_diff_sum / len(prices)
        volatility = math.sqrt(mean_squared_diff)
        print(f"Volatility of {window} is {volatility}")
```

[11]:
```python
parameters = {"ma": 5, "volatility": 4}
find_indicators(prices, **parameters)
```

```
Moving Average (5) is 23.6
Volatility of 4 is 18.191687662226393
```

1.5 5. 导入函数 Import function

将函数 find_indicators 保存到名为 function.py 的文件内。

通过导入模块的方式，把 function.py 文件内容导入到当前程序，实现调用不同.py 文件的函数功能。

[12]:
```python
import function
function.find_indicators(prices, **parameters)
```

```
Moving Average (5) is 23.6
Volatility of 4 is 18.191687662226393
```

[13]:
```
# 给导入的模块起别名
import function as f
f.find_indicators(prices, **parameters)
```

```
Moving Average (5) is 23.6
Volatility of 4 is 18.191687662226393
```

[14]:
```
# 导入模块内特定的函数
from function import find_indicators
find_indicators(prices, **parameters)
```

```
Moving Average (5) is 23.6
Volatility of 4 is 18.191687662226393
```

[15]:
```
# 导入模块内所有的函数
# 不推荐该用法，假设模块内有大量函数，容易和主程序出现函数或者变量名重复。

from function import *
find_indicators(prices, **parameters)
```

```
Moving Average (5) is 23.6
Volatility of 4 is 18.191687662226393
```

1.6 6. 将函数作为对象使用，Function as objects

程序化交易中，交易所对期货合约代码有各自的规范要求。

其中上期所、上期所能源中心和大商所要求合约代码字母小写，

中金所和郑商所要求合约代码字母大写，

且郑商所对合约数字部分要求为 3 位，其余交易所要求为 4 位。

以上要求导致程序化交易过程中，需要对期货合约名称进行清洗，以符合规范。

下方用代码简单演示过程。

[16]:
```
# 根据以上要求，先对合约字母部分作清洗，然后清理郑商所合约的数字长度。

import re # 正则表达式模块，针对字符串的识别、修改。
```

```
futures_contracts = ["RB2310.shfe", "m2309.DCE", "Ic2309.Cffex", "sr2309.CZCE",␣
↪"LU2309.INE"]

def clean_symbol(contract):
        uppercase_exchange = ["CZCE", "CFFEX"]
        lowercase_exchange = ["SHFE", "DCE", "INE"]
        symbol, exchange = contract.split(".")
        if exchange.upper() in uppercase_exchange:
                contract = ".".join([symbol.upper(), exchange.upper()])
        elif exchange.upper() in lowercase_exchange:
                contract = ".".join([symbol.lower(), exchange.upper()])
        return contract

def clean_CZCE(contract):
        symbol, exchange = contract.split(".")
        if exchange == "CZCE":
                # 对于郑商所的期货品种，识别保留其字母部分，取数字部分的后三位拼接。
                symbol = re.findall(r"[A-Za-z]+", symbol)[0] + re.
↪findall(r"\d{3}$", symbol)[0]
                contract = ".".join([symbol, exchange])
        return contract
```

[17]:
```
for i in futures_contracts:
        print(clean_CZCE(clean_symbol(i)))
```

```
rb2310.SHFE
m2309.DCE
IC2309.CFFEX
SR309.CZCE
lu2309.INE
```

[18]:
```
# 将函数作为对象遍历
# 这么做的好处是能够更加清楚地看到合约清洗的过程步骤，且更加方便后期维护
clean_processes = [clean_symbol, clean_CZCE]
for i in futures_contracts:
    for process in clean_processes:
        i = process(i)
```

```
    print(i)
```

```
rb2310.SHFE
m2309.DCE
IC2309.CFFEX
SR309.CZCE
lu2309.INE
```

1.7 7. Generator 生成器

函数在运行时会事先将所有的值存储在内存中，如果 prices 是一个非常大的列表，那么运行 find_rolling_mean 将会占用大量资源。

python 有一种特殊类型的函数，Generator 生成器，它可以通过返回一个迭代器的方式，避免计算资源的大量使用。

生成器函数被定义之后，可以用 next() 手动获取生成器的值，也可以用 for 循环遍历。

将 find_rolling_mean 改写成一个生成器作例子。删除 function 内部的列表，将 return 改写成 yield。

[19]:
```
def rolling_means_generator(prices: list, window_size: int):
    for i in range(len(prices) - window_size + 1):
        price_piece = prices[i:i+window_size]
        mean = sum(price_piece) / window_size

        yield mean
```

[20]:
```
# 手动获取生成器的值
generator = rolling_means_generator(prices, window_size)
next(generator)
```

[20]: 8.6

[21]:
```
next(generator)
```

[21]: 15.0

然而，当 generator 遍历完，没有更多值可生成时，会引发 StopIteration 异常

为了避免出现这种问题，可以使用 for 循环代替

[22]:
```
# for 循环会接上之前的结果。
for mean in generator:
    print(mean)
```

```
15.6
23.6
```

生成器还可以用表达式定义 Generator Expression。
和列表解析式的区别就在于，GE 用的是小括号。

[23]:
```
rolling_means_generator2 = (
    (lambda price_piece: sum(price_piece) / window_size)(prices[i:␣
 ↪i+window_size])\
  for i in range(len(prices) - window_size + 1)
)
```

[24]:
```
next(rolling_means_generator2)
```

[24]:
```
8.6
```

1.8 8. 异常处理

[25]:
```
# 使用 next() 访问生成器，在一次遍历结束后会报错，
# 为了防止 StopIteration 出现，除了 for 循环，还可以用 try-except 组合，对报错情况
进行处理

generator = rolling_means_generator(prices, window_size)

try:
    while True:
        mean = next(generator)
        print(mean)
except StopIteration: # 如果出现遍历完成之后的报错
    print("Iteration terminated")
    pass
```

```
8.6
15.0
15.6
```

```
23.6
Iteration terminated
```

2 Summary

- 函数简介
- lambda 匿名函数
- 任意数量参数 *variables
- 任意数量键参数 **kwargs
- 导入模块 import function
- 函数作为对象使用
- 生成器 Generator
- 异常处理 try except