金融计算机第五周作业

——Part1 与 Part2

一、常用的对象检视函数和语句

1.创建第五周打卡仓库，并 Clone

```
(base) PC@DESKTOP-FTJ84MN MINGW64 ~
$ git config --global user.name Leo_Sunny
git config --global user.email zhuyj_8219@163.com

(base) PC@DESKTOP-FTJ84MN MINGW64 ~
$ cd repo

(base) PC@DESKTOP-FTJ84MN MINGW64 ~/repo
$ ls -l
total 20
drwxr-xr-x 1 PC 197121   0  3月 18 16:47 Mywork/
-rw-r--r-- 1 PC 197121 981  3月 12 07:45 script1.py
drwxr-xr-x 1 PC 197121   0  3月 13 18:12 week01/
drwxr-xr-x 1 PC 197121   0  3月 18 18:07 week02/
drwxr-xr-x 1 PC 197121   0  3月 26 00:47 week03/
drwxr-xr-x 1 PC 197121   0  4月  2 09:20 week04/

(base) PC@DESKTOP-FTJ84MN MINGW64 ~/repo
$ pwd
/c/Users/PC/repo

(base) PC@DESKTOP-FTJ84MN MINGW64 ~/repo
$ git clone git@gitcode.com:Leo_Sunny/week05.git
Cloning into 'week05'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 5 (delta 0), pack-reused 0 (
Receiving objects: 100% (5/5), 8.44 KiB | 720.00 KiB/s, done
```

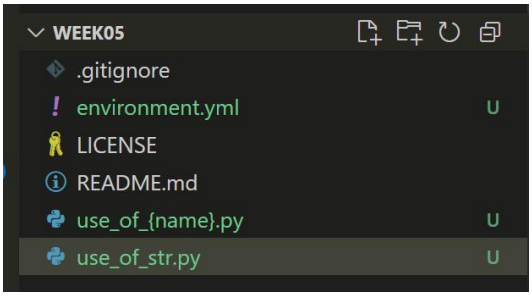
这里 clone 的是 SSH 下的克隆代码，而不是 https 下的克隆代码

```
(base) PC@DESKTOP-FTJ84MN MINGW64 ~/repo
$ cat week04/environment.yml
name: week04
channels:
  - conda-forge
dependencies:
  - python=3.12
  - wat-inspector
(base) PC@DESKTOP-FTJ84MN MINGW64 ~/repo
$ cp week04/environment.yml week05/

(base) PC@DESKTOP-FTJ84MN MINGW64 ~/repo
$ ls -l week05
total 25
-rw-r--r-- 1 PC 197121    91  4月  7 20:21 environment.yml
-rw-r--r-- 1 PC 197121 18805  4月  7 20:20 LICENSE
-rw-r--r-- 1 PC 197121  2239  4月  7 20:20 README.md
```

通过复制的方法在 week05 文件夹中创建 environment.yml 文件，并使用代码：
conda env create 创建 conda 环境
（在此之前修改复制好后的 env 文件中的 name 为 week05，如下图所示）

```
environment.yml
1  name: week05
2  channels:
3    - conda-forge
4  dependencies:
5    - python=3.12
6    - wat-inspector
```



在 code 中创建一个新的.py 文件：
use_of_{name}.py，其中 name 替换为要创建的对象类型如 use_of_str.py、use_of_bytes.py

```
(base) PC@DESKTOP-FTJ84MN MIN
$ conda activate week05
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/
$ python use_of_str.py
(week05)
```

激活环境并使用终端打开 python 文件
发现文件内什么也没有，但本身其实是有代码：**a = 'hello'**

```
use_of_str.py > ...
1   a = "hello"
2   print(a)
3
```

所以需要在第二行加入代码：print(a)
再进行文件打开，即可看到内容了，如下图

```
(week05)
PC@DESKTOP-FTJ84MN MINGW6
$ python use_of_str.py
hello
```

2.
(1). **id()** --返回对象在**虚拟**内存中的**地址**(正整数)，如果 id(a) == id(b)，那么 a is b(is 是个运算符)

```
use_of_str.py > ...
1   a = "hello"
2   x = id(a)
3   print(x)
```

修改代码后在终端运行，结果如下图

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/
$  python use_of_str.py
1732224113600
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/
$  python use_of_str.py
1544600378304
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/
$  python use_of_str.py
2481093434304
```

发现每次运行结果不一样，因为 id 输出的是一个**虚拟地址**

```
use_of_str.py > ...
1   a = "hello"
2   b = 'hello'
3   x = id(a)
4   print(x)
5   y = id(b)
6   print(y)
7
```

修改代码验证**两个值一样的变量的虚拟地址是否一样**

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/w
$ python use_of_str.py
2105513630656
2105513630656
```

如上图所示，显示的虚拟地址是一样的

```
use_of_str.py > ...
1   a = [2,5]
2   b = [2,5]
3   x = id(a)
4   print(x)
5   y = id(b)
6   print(y)
7
```

修改代码，验证猜想：地址是否一样？

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/
$ python use_of_str.py
2440100780288
2440100778304
```

虚拟地址不同：因为对象不同，虽然内容等同但是，不是同一个对象

```
use_of_str.py > ...
1   a = [2, 5]
2   b = [2, 5]
3   x = id(a)
4   print(x)
5   y = id(b)
6   print(y)
7   a[0] = 9
8   print(a)
9   print(b)
10  print(id(a))
11  print(id(b))
```

修改代码如上图所示，运行结果如下图所示：

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week05 (mai
$ python use_of_str.py
1937677752576
1937677750592
[9, 5]
[2, 5]
1937677752576
1937677750592
```

得出结论：

print(id(a))=print(x)

print(id(b))=print(y)

(2)type()--返回对象的类型　与　isinstance()--判断对象是否属于某个(或某些)类型

```
use_of_str.py > ...
1   a = [2, 5]
2   b = [2, 5]
3   x = id(a)
4   print(x)
5   y = id(b)
6   print(y)
7   a[0] = 9
8   print(a)
9   print(b)
10  print(id(a))
11  print(id(b))
12  print(type(a))
13  print(isinstance(a, str))
```

修改代码如上图所示，运行结果如下图所示：

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week05 (main)
$ python use_of_str.py
2521332979968
2521332977984
[9, 5]
[2, 5]
2521332979968
2521332977984
<class 'list'>
False
```

(3)dir()--返回对象所支持的属性(attributes)的名称列表

(4)str()--返回对象 print 时要显示在终端的字符串

```
print(id(b))
print(type(a))
print(isinstance(a, str))
print(isinstance(a, list))
print("dir(a):", dir(a))
```

```
2541418846464
2541418844480
<class 'list'>
False
True
dir(a): ['__add__', '__class__
, '__eq__', '__format__', '__g
 '__imul__', '__init__', '__in
```

根据 isinstance 函数验证变量的类型，是 list

```
print(isinstance(a, str))
print(isinstance(a, list))
print(isinstance(a, (str, list)))
print("dir(a):", dir(a))
```

这个意思是如果是 list 或者 str 其中一个那么就返回 True，运行结果如下图

```
2090389344576
<class 'list'>
False
True
True
dir(a): ['__add__', '_
```

如果都不是那就返回 False

(5)print()函数将表达式（expression）输出到终端，查看结果是否符合预期

(6)assert 语句查验某个表达式（expression）为真，否则报错（AssertionError）退出

```
print(isinstance(a, str))
print(isinstance(a, list))
print(isinstance(a, (str, list)))
print("dir(a):", dir(a))
assert isinstance(a, str)
```

```
True
True
dir(a): ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__
, '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate_
 '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr
tr__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'sort']
Traceback (most recent call last):
  File "C:\Users\PC\repo\week05\use_of_str.py", line 17, in <module>
    assert isinstance(a, str)
           ^^^^^^^^^^^^^^^^^^
AssertionError
```

如上图所示，在第 17 行运行后产生报错并退出运行

查验是否真的退出运行：

添加代码 print(“apple”)，运行结果发现后面的 apple 没有显示，说明确实推出运行

```
print("dir(a):", dir(a))
assert isinstance(a, str)
print("apple")
```

```
True
True
dir(a): ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__
, '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate_
 '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__l
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr_
tr__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', '
'sort']
Traceback (most recent call last):
  File "C:\Users\PC\repo\week05\use_of_str.py", line 17, in <module>
    assert isinstance(a, str)
           ^^^^^^^^^^^^^^^^^^
AssertionError
(week05)
```

将 str 修改为 list，再进行运行，apple 即可运行

```
print(isinstance(a, (str, list)))
print("dir(a):", dir(a))
assert isinstance(a, list)
print("apple")
```

```
True
True
dir(a): ['__add__', '__class__', '__class_getitem__'
, '__eq__', '__format__', '__ge__', '__getattribute_
 '__imul__', '__init__', '__init_subclass__', '__ite
 '__reduce__', '__reduce_ex__', '__repr__', '__rever
tr__', '__subclasshook__', 'append', 'clear', 'copy'
'sort']
apple
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week05 (main)
```

(7)try 语句拦截报错，避免退出，将流程（flow）转入 except 语句

```
print("dir(a):", dir(a))
try:
    assert isinstance(a, str)
except AssertionError:
    print("type error")
print("apple")
```

展示 try 语句的作用，代码如上图所示，运行结果如下图所示，使不报错，可继续运行

```
True
dir(a): ['__add__', '__class__', '__class_g
, '__eq__', '__format__', '__ge__', '__geta
 '__imul__', '__init__', '__init_subclass__
 '__reduce__', '__reduce_ex__', '__repr__',
tr__', '__subclasshook__', 'append', 'clear
'sort']
apple
```

(8)breakpoint()函数暂停程序运行，进入 pdb 调试 (debug) 模式

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week05 (main)
$ python -m pdb use_of_str.py
> c:\users\pc\repo\week05\use_of_str.py(1)<module>()
-> a = [2, 5]
(Pdb) l
  1  -> a = [2, 5]
  2     b = [2, 5]
  3     x = id(a)
  4     print(x)
  5     y = id(b)
  6     print(y)
  7     a[0] = 9
  8     print(a)
  9     print(b)
 10     print(id(a))
 11     print(id(b))
(Pdb)
```

回顾使用调试器的方法与代码

```
try:
    assert isinstance(a, str)
except AssertionError:
    breakpoint()
    print("type error")
print("apple")
```

在代码的某个节点使用函数 breakpoint()，可以召唤出调试器，并运行到此处暂时停止

```
False
True
True
dir(a): ['__add__', '__class__', '_
, '__eq__', '__format__', '__ge__',
 '__imul__', '__init__', '__init_su
 '__reduce__', '__reduce_ex__', '__
tr__', '__subclasshook__', 'append'
'sort']
> c:\users\pc\repo\week05\use_of_st
-> print("type error")
(Pdb)
```

```
-> print("type error")
(Pdb) l
 16     print("dir(a):", dir(a))
 17     try:
 18         assert isinstance(a, str)
 19     except AssertionError:
 20         breakpoint()
 21  ->     print("type error")
 22     print("apple")
[EOF]
```

即将运行第 21 行，停在了这里

二、获得 str 类型实例的几种途径

1.通过表达式得到实例（实例化）

!什么是实例

实例：实际的例子

(1)字面值

代码中直接表示特定数据值的常量，它以原始形式出现，无需通过计算或变量引用获取。包括“数值字面值”、“字符字面值”、“字符串字面值”、“布尔字面值”、“其他字面值”

以下是举例说明：

①str（在 str 文件中的操作）

新建一个文件夹，命名为：**use_of_str_2.py**

编写代码如下图所示

```
use_of_str_2.py > ...
1  print("字面值")
2  s = "university"
3  print(isinstance(s, str))
4  assert type(s) is str
5
```

运行结果如下图所示

```
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/r
$ python use_of_str_2.py
字面值
True
```

创建新的代码

其中：**-f 字符串**（也称为格式化字符串字面值）。f 是一个前缀，用于告诉 Python 这是一个格式化字符串。在花括号{}中嵌入表达式，Python 会在运行时自动将这些表达式的值插入到字符串中。

\t 是一个转义序列，代表制表符（Tab）。在字符串中使用\t 时，Python 会将其解释为

一个制表符，用于在输出中创建水平间距，实现文本的对齐效果。

\n 也是一个转义序列，代表换行符。当在字符串中使用\n 时，Python 会将其解释为一个换行操作。

```
print("f-string")
x = "Tom"
s = f"name: {x}"
print(s)

s = "a\tb"
print("TAB", s)

s = "aaa\nbbb"
print("New Line", s)
```

代码输出结果如下图所示：

```
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week05 (main)
$ python use_of_str_2.py
字面值
university
True
f-string
name: Tom
TAB a   b
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week05 (main)
$ python use_of_str_2.py
字面值
university
True
f-string
name: Tom
TAB a   b
New Line aaa
bbb
```

撰写新代码如下：

在 Python 中，使用三个双引号""" """或三个单引号''' '''表示多行字符串，Python 会按照字符串中换行符等符号的位置，逐行输出字符串内容

```
s = """xyz
abc
  eee
aaa
"""
print(s)
```

输出结果如下图所示

```
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/week
$ python use_of_str_2.py
字面值
university
True
f-string
name: Tom
TAB a   b
New Line aaa
bbb
xyz
abc
  eee
aaa
```

(2)推导式

在字符串里面没有推导式

推导式只适用于列表、字典、集合

(3)初始化

在使用变量、对象或数据结构之前，为其赋予初始状态 。

包括“变量初始化”、“对象初始化”、“数据结构初始化”

以下是举例说明：

①str（在 str 文件中的操作）

```
print("初始化")
s = str()
print(s)
```

在 s 中尝试什么都不输入，只放一个字符串初始化函数，检验是否能够运行，运行结果如下图所示、运行结论为**可以运行**

在这里的 str()的意思是代表空字符串

```
xyz
abc
  eee
aaa

初始化

(week05)
```

重新为 s 进行赋值，并输出 print

```
print("初始化")
s = str()
print(s)
s = str([5, 8, 2])
print(s)

assert str([5, 8, 2]) == "[5, 8, 2]"
```

运行结果如下图所示

```
初始化

[5, 8, 2]
(week05)
```

上述的 assert 语句也没有报错，所以说明初始化后的字符串和字符串本身是相同的

```
assert str([5, 8, 2]) == "[5, 8, 2]"
assert str(1.1 + 2.2) == "3.3"
```

```
初始化

[5, 8, 2]
Traceback (most recent call last):
  File "C:\Users\PC\repo\week05\use_of_str_2.py", line 35, in <module>
    assert str(1.1 + 2.2) == "3.3"
           ^^^^^^^^^^^^^^^^^^^^^^^
AssertionError
(week05)
```

在第二个 assert 代码运行后，程序报错，想要查找原因需要开启调试器 pdb，使用代码**“c”**直接运行到报错为止，得到如下图的结果。

```
AssertionError
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> c:\users\pc\repo\week05\use_of_str_2.py(35)<module>()
-> assert str(1.1 + 2.2) == "3.3"
(Pdb)
```

```
-> assert str(1.1 + 2.2) ==
(Pdb) p str(1.1+2.2)
'3.3000000000000003'
(Pdb)
```

在 pdb 调试器下使用 p 代码，输出 print 报错那一行其中一个表达式，得到上面的结果，发现与另一表达式内容不同，说明二者不等。

```
33
34  assert str([5, 8, 2]) == "[5, 8, 2]"
35  assert str(1.1 + 2.2) != "3.3"
36
```

将代码修改为不等再次运行，发现可以运行了，说明代码修改正确，如下图所示

```
初始化

[5, 8, 2]
(week05)
PC@DESKTOP-FTJ84MN MINGW
```

(4)运算值

运算值指参与运算操作的数据值

以下是举例说明：

①str（在 str 文件中的操作）

```
36
37  s = "="
38  s = s * 20
39  print(s)
40
```

s 是新生成了一个长度为 20 的字符串

检验方法如下：

1.首先在代买前面一行添加断点 breakpoint()

而后在调试器 pdb 中，按行运行

由运行可知，两个 s 的 id 不同，所以不是同一个 s，不是同一个内容

```
 38     s = "="
 39  -> s = s * 20
 40     print(s)
[EOF]
(Pdb) p s
'='
(Pdb) id(s)
140717891946368
(Pdb) n
> c:\users\pc\repo\week05\use_of_str_2.py(40)<module>()
-> print(s)
(Pdb) p s
'===================='
(Pdb) l
 35     assert str(1.1 + 2.2) != "3.3"
 36
 37     breakpoint()
 38     s = "="
 39     s = s * 20
 40  -> print(s)
[EOF]
(Pdb) id(s)
2464802373040
```

2.方法 2

```
37  s = "="
38  x = id(s)
39  s = s * 20
40  y = id(s)
41  print(s)
42  assert x != y
43
```

```
初始化

[5, 8, 2]
====================
The program finished and will be restarted
> c:\users\pc\repo\week05\use_of_str_2.py(1)<module>()
-> print("字面值")
(Pdb)
```

在代码中记录 id，并使用 assert 语句查看检验 x 与 y 的关系

运行结果不报错，说明 x 与 y 确实不相等

```
43
44  s = "hello"
45  assert s[3] == "l"
46
```

```
[5, 8, 2]
====================
The program finished and will be restarted
> c:\users\pc\repo\week05\use_of_str_2.py(1)<module>()
-> print("字面值")
```

(5)索引值

索引值用于标识序列（如字符串、列表、元组等 ）中元素的位置，从 0 开始计数；

负向索引：从序列末尾开始计数，-1 表示最后一个元素

多维索引：对于二维或更高维的序列（如二维列表 ），需多个索引值定位元素

以下是举例说明：

①str（在 str 文件中的操作）

```
s = "hello"
assert s[3] == "l"
assert s[-1] == "o"
assert s[:3] == "hel"
assert s[4] == s[-1]
try:
    s[5]
except IndexError as e:
    print(e)
```

```
xyz
abc
  eee
aaa

初始化

[5, 8, 2]
====================
The program finished and will be restarted
> c:\users\pc\repo\week05\use_of_str_2.py(1)<module>()
-> print("字面值")
(Pdb)
```

说明所有色 assert 语句都能通过并且没有输出 e，说明 try 语句 **s[5]**没有语法错误

(6)返回值

返回值是函数执行完毕后返回给调用者的结果。

```
s = "hello"
u = s.upper()
print(u)
print(s)
```

```
[5, 8, 2]
====================
string index out of range
HELLO
hello
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo
```

上述案例中 upper 使所有字符都大写的函数

```
t = "name:{},age {}"
print(t)
t1 = t.format("Jack", 21)
print(t1)
```

```
HELLO
hello
name:{},age {}
name:Jack,age 21
(week05)
```

(7)运算符

1.+——支持

```
s1 = "abc"
s2 = "ghi"
s = s1 + s2
assert s == "abcghi"
print(s1 + s2)
```

```
string index out of range
HELLO
hello
name:{},age {}
name:Jack,age 21
abcghi
(week05)
PC@DESKTOP-FTJ84MN MINGW64 ~/repo/w
```

2.-——不支持

```
print(s1 + s2)
print(s2 - s1)
```

```
abcghi
Traceback (most recent call last):
  File "C:\Users\PC\repo\week05\use_of_str_2.py", line 69
    print(s2 - s1)
          ~~~^~~~
TypeError: unsupported operand type(s) for -: 'str' and '
(week05)
```

```
s = "=*="
s = s * 10
print(s)
```

```
name:Jack,age 21
abcghi
unsupported operand type(s) for -: 'str' and 'str'
=*==*==*==*==*==*==*==*==*=
(week05)
```