9.1 今日目标

9.2【记忆】property属性

- 介绍：

  > 可以把方法当做属性来访问。

- **装饰器方式**：
  - 获取属性 `@property`
  - 设置属性 `@方法名.setter`

```
class Person(object):

 def __init__(self):
     self.__age = 0

 # 在只读的方法上，加装饰器@property  ，表示下面的方法是属性(get方法)
 @property
 def age(self):
     """提供公有方法，访问私有属性"""
     return self.__age

 # 在另一个同名的方法上，加装饰器 @方法名.setter，表示下面的方法是属性(set方法)
 @age.setter
 def age(self, age):
     """提供公有方法，设置私有属性"""
     if 0 <= age <= 100:
         self.__age = age
     else:
         print("年龄需要在0~100之间")

# 此时，就可以使用属性的方式，来设置值和获取值
# 以后使用时，就会变得很方便
p = Person()
p.age = 123456
print(p.age)
```

- 类属性方式(了解)：`属性 = property(属性的get方法，属性的set方法)`

```
class Person(object):

 def __init__(self):
     self.__age = 0

 def get_age(self):
     """提供公有方法，访问私有属性"""
```

```
        return self.__age

    def set_age(self, age):
        """提供公有方法，设置私有属性"""
        if 0 <= age <= 100:
            self.__age = age
        else:
            print("年龄需要在0~100之间")

    # 类属性：内存只存在一份，类的所有对象公用
    # property实际上就是一个类. 第一个参数必须是属性的get方法，第二个参数必须是属性的set方法
    age = property(get_age, set_age)


p = Person()
p.age = 999
print(p.age)
```

9.3【理解】with语句和上下文管理器

- with语句的使用: 必须和上下文管理器配置使用

```
# with 上下文管理器:open("1.txt", "w")
with open("1.txt", "w") as f:
    pass
```

- 上下文管理器类:
  - 必须实现两个方法:

    1. `__enter__`
    2. `__exit__` 在退出的方法中, 进行资源的释放

- 上下文管理器执行顺序

    1. with语句开始时候，会自动调用上文函数
    2. with语句结束的时候，会自动调用下文函数

```
class File(object):

    def __init__(self, filename, filemode):
        self.filename = filename
        self.filemode = filemode

    def __enter__(self):
        """上文的方法"""
        print("上文的方法")
        self.file = open(self.filename, self.filemode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        """下文的方法"""
        # 这里是核心，用于资源的释放
        print("下文的方法")
```

```
        self.file.close()


# 创建上下文管理器对象
file = File("1.txt", "w")

# 使用with和上下文管理器连接
# with和上下文管理器对象连用时，会自动调用__enter__方法
# 当with缩进执行完毕，会自动调用__exit__方法
with File("1.txt", "w") as f:
    pass
```

9.4【记忆】生成器的创建方式

- 介绍：生成器记录的是算法, 不是一次生成所有数据，而是使用一个生成一个，节约内存空间。
- 使用方式：
  - 生成器推导式：

```
# 创建生成器：把列表推导式的[]改成()
generator1 = (i * 2 for i in range(5))

# 获取数据  next(生成器)

"""使用for循环获取生成器的数据"""
# for循环的底层原理：先问传入的数据，能否取出下一个数据(这时就会调用next()方法)，能取出就赋值给前面的临时变量. 如果不能取出，就终止
# for循环会自动调用next()，取不出来，自动调用break
for data in generator1:
 print(data)
```

  - yield关键字

```
def mygenerater(n):
 for i in range(n):
     yield i

g = mygenerater(2)
for data in g:
 print(data)
```

- 获取生成器数据
  - ``获取下一个数据
  - `` 获取后面所有数据
- 生成器的使用场景：
  - 斐波拉契数列（Fibonacci）

> 第0次: result=0 a=0 b=1
>
> 第1次: result=0 a=1 b=1
>
> 第2次: result=1 a=1 b=2

> 第3次: result=1 a=2 b=3
>
> 第4次: result=2 a=3 b=5
>
> 第5次: result=3 a=5 b=8
>
> 第6次: result=5 a=8 b=13

```python
def fibonacci(num):
    a = 0   # 2   3
    b = 1   # 3   5

    # 记录生成fibonacci数字的下标
    current_index = 0

    # result: 会先记录原本的a的值
    # 记录完成之后，就会让a和b发生变化
    # 然后返回原本的a的值(result)
    # 当下一次循环进来时，会让result再次记录新的a的值
    while current_index < num:
        result = a  # result = 2
        a, b = b, a + b
        current_index += 1
        # 代码执行到yield会暂停，然后把结果返回出去，下次启动生成器会在暂停的位置继续往下执行
        yield result


fib = fibonacci(10)
# 遍历生成的数据
for value in fib:
    print(value)
```

9.5 【理解】深拷贝和浅拷贝

- 浅拷贝：
  - 不可变类型的浅拷贝: 是引用，不会拷贝
  - 可变类型的浅拷贝: 看第一层，最多也就拷贝一层
- 深拷贝：
  - 不可变类型的深拷贝: 拷贝到有可变类型的那一层
  - 可变类型的深拷贝：拷贝到有可变类型的那一层
- 浅拷贝和深拷贝的区别
  - 浅拷贝最多拷贝对象的一层
  - 深拷贝可能拷贝对象的多层

9.6【了解】正则表达式概述

- 正则表达式概念：符合某种规则的字符串的代码
- 正则表达式的作用：匹配、查找、验证符合某种规则的字符串
- re模块的作用： 在python编程中需要使用正则表达式，需要导入re模块

- re模块的使用步骤：
  - 导入模块

    ```
    import re
    ```

  - 使用match() 方法进行检测

    ```
    match(正则表达式，要匹配的目标字符串)
    ```

  - 判断是否检测/匹配成功

    ```
    if result:
    ```

  - 取出匹配的具体内容

    ```
    result.group()
    ```

9.7【记忆】匹配单个字符

- `.`： 任意一个字符（除了\n)
- `[]`: 匹配[]中列举的任意一个字符
- `\d`: 匹配0-9任意一个字符
- `\D`: 非0-9的任意一个字符
- `\s`: 匹配空白，空格，tab键，\r, \n
- `\S`: 匹配非空白
- `\w`: 匹配非特殊字符, a-z, A-Z, 0-9, _, 汉字.
- `\W`: 匹配特殊字符. 非字母，非数字，非_, 非汉字

9.8【记忆】匹配多个字符

- - `: 匹配前一个字符出现0次或者无限次.
- `+`: 匹配前一个字符出现1次或者无限次.
- `?`： 匹配前一个字符出现0次或者1次.
- `{m}`： 匹配前一个字符出现m次.
- `{m,n}`: 匹配前一个字符出现最少m次，最多n次.

  > `{m,}`: 匹配前一个字符出现最少m次.
  >
  > `{,n}`: 匹配前一个字符出现最多n次

9.9【记忆】匹配开头结尾

- `^` 表示 匹配 以后一个字符开头

  > ^ 有两个作用：
  >
  > 1）^\d : 表示以数字开头

  ```
  result = re.match("^1\d{10}", "17712345678")
  ```

> 2) `[^ETM]`：表示除了`ETM`以外字母的都匹配
>
> ```
> result = re.match("C[^ETM]O", "CAO")
> ```

- `$`表示 匹配 以前一个字符结尾

> \d$ 表示数字结尾
>
> ```
> result = re.match(".+\d$", "12")
> ```

9.10【理解】匹配分组

- `|`的作用：匹配左边或者右边的正则表达式

```
my_list = ["apple", "banana", "orange", "pear"]

for item in my_list:
    result = re.match("apple|pear", item)
    if result:
        print("匹配成功，提取匹配的字符串：", result.group())
    else:
        print("匹配失败")
```

- 分组，整体匹配:`()`：把括号的数据作为分组匹配

```
# (ab)  将括号中字符作为一个分组
# 需求：匹配出163、126、qq等邮箱
# \转义符  . 原本的.有特殊意义. 再加一个\，表示取消了特殊的意义
result = re.match("[a-zA-Z0-9_]{4,20}@(163|126|qq)\.(com|cn)", "abcd@126.cn")

# 需求：匹配qq:10567这样的数据，提取出来qq文字和qq号码
# ()：第一个()中匹配的内容就是第一分组，group(1)
#     第二个()中匹配的内容就是第二分组，group(2)
result = re.match("(qq):([^0]\d{4,10})", "qq:10567")
```

```
# 提取子字符串
if result:
    print("匹配成功，提取匹配的字符串：", result.group())
    print("匹配成功，提取匹配的字符串：", result.group(1))
    print("匹配成功，提取匹配的字符串：", result.group(2))
else:
    print("匹配失败")
```

- 引用 分组

  \1 表示引用第一组

```
result = re.match("<([a-zA-Z1-6]+)><([a-zA-Z1-6]+)>.*</\\2></\\1>", "<html>
<h1>www.itcast.cn</h1></html>")
if result:
    print("匹配成功，提取匹配的字符串：", result.group())
    print("匹配成功，提取匹配的字符串：", result.group(1))
    print("匹配成功，提取匹配的字符串：", result.group(2))
else:
    print("匹配失败"
```

> `\\1` 表示引用第一组，`\\`是转义字符，转义后代表一个 `\`
>
> `\\2` 表示引用第二组

- 分组起别名
  - 起名

    ```
    (?P<name>)  分组起别名
    ```

  - 引用别名

    ```
    (?P=name)  引用别名为name分组匹配到的字符串
    ```

  - 整体代码：

    ```
    # 需求：匹配出<html><h1>www.itcast.cn</h1></html>
    # result = re.match("<(?P<name1>[a-zA-Z1-6]+)><(?P<name2>[a-zA-Z1-6]+)>.*</\\2>
    </\\1>", "<html><h1>www.itcast.cn</h1></html>")
    result = re.match("<(?P<name1>[a-zA-Z1-6]+)><(?P<name2>[a-zA-Z1-6]+)>.*</(?
    P=name2)></(?P=name1)>", "<html><h1>www.itcast.cn</h1></html>")
    if result:
        print("匹配成功，提取匹配的字符串：", result.group())
        print("匹配成功，提取匹配的字符串：", result.group(1))
        print("匹配成功，提取匹配的字符串：", result.group(2))
    else:
        print("匹配失败")
    ```

9.11 今日内容总结