欢迎使用Markdown

> 记录一些python的杂例. 目前学习路线是计算机的第0课+笨办法+速成课+cs61a.
>
>

Lec 10. Containers

Lists, Ranges, Strings

- **Lists** is **built-in** data type in python. Lists contain other values.
- **Ranges** 是另一种 sequence type, **不是** lists, 是一列连续的整数.
- Sequence Processing (For statement)
- **Strings** 是Abstraction, 是文本数据的表示 (不关心如何编码)

```
# 列表的乘法
>>> [1, 8, 2, 8 ] * 2
[1, 8, 2, 8, 1, 8, 2, 8]

# 将 range 转化成列表
>>> list(range(4))
[0, 1, 2, 3]

for _ in range(2, 7):
    print(_)
# 输出: 2 3 4 5 6, 没有7, 不需要关心_是什么, 也就是2到7-1

>>> city = 'Berkeley'
>>> city[3]
'k'
>>> 'here' in "Where's Waldo"
True
```

- For statement 为迭代序列而生.

```
for <name> in <expression>:
    <suite> # 程序

def count(s, value):
    """Count the number of times that value in sequence s.
    >>> count([1, 2, 1, 2, 1], 1)
    3
    """
    total = 0
    for _ in s:
        if _ == value:
            total += 1
    return total
# 列表才有 [_ for _ in s if _ == value]

# x 就是列表中的元素, 这是 for 语句特性(do iteration)
>>> odds = [1, 3, 5, 7, 9]
>>> [x+1 for x in odds]
[2, 4, 6, 8, 10]
```

```
# Sequence Unpacking
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
>>> for x, y in pairs:
...     if x == y:
...         same_count += 1
```

```
# 计算 List L 中的元素之和
def mysum(L):
    sum = 0
    for _ in L:
        sum+= _
    return sum
```

```
def mysum(L):
    sum = 0
    if len(L) == 0:
        return 0
    else:
        return L[0] + mysum(L[1:]) # 切片去除第一个元素
```

```
# 计算 1 + 2 + ... + n, n 为正整数
def sum_rec(n):
    if n == 0:
        return 0
    else:
        return n + sum_rec(n-1)
```

```
def sum_iter(n):
    sum = 0
    for _ in range(n+1):
        sum += _
    return sum
```

```
# Reversing a list (recursively)
reverse("ward") = reverse("ard") + "w"
```

```
def reverse(s):
    if len(s) <= 1:
        return s
    else:
        return reverse(s[1:]) + s[0]
```

```
# 切片留下前2个元素
>>> [1, 2, 8, 8][:2]
[1, 2]

# 不能直接 nums = new_list，而必须用 nums[:] = new_list.
nums = [1, 2, 3]
ref = nums  # `ref` 现在指向 `nums`
nums = [4, 5, 6]  # nums 现在指向新列表，但 `ref` 仍指向旧列表
print(ref)  # [1, 2, 3] ❌ 没有同步更新
print(nums)  # [4, 5, 6] ✅ `nums` 本身变了
```

介绍一些函数

```
l1 = list('I love Python')#字符串为可迭代类型
>>> print(l1)
['I', ' ', 'l', 'o', 'v', 'e', ' ', 'P', 'y', 't', 'h', 'o', 'n']

l2 = list([1,'dormitory',[{12,456},'忙']])#列表为可迭代类型
>>> print(l2)
[1, 'dormitory', [{456, 12}, '忙']]
```

```
remove
```

```
# pop 删除第 i 个位置的元素
hand = ['A', 'K', 'Q', 'J', 10, 9]
>>> hand.pop(1)
'K'
>>> hand
['A', 'Q', 'J', 10, 9]
```

Lec 11. Data Abstract

1. 抽象的数据类型将复合的对象视为一个整体进行操作，是隔开了representation和use的方法. 例如一条直线 f,

- 使用 `slope(f)` 代替 `f[0]`
- 使用 `y_intercept(f)` 代替 `f[1]`
  这样子代码更易读、易修改，无需改动 `f` 的代码
- 有理数

$$\frac{numerator}{denominator}$$

- Constructor: rational(n, d): 返回有理数
- Selector:
  - numer(x): 返回分子
  - denom(x): 返回分母

```
# 第一层，保证有理数表示的唯一性
from fractions import gcd # python3.9 变为 from math import gcd
def rational(n, d):
    g = gcd(n, d)
    return [n//g, d//g]

# 返回分子
def numer(x):
    return x[0]

# 返回分母
def denom(x):
    return x[1]

# 第二层，利用Constructor和Selector实现有理数的加法
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)

def divide_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy, dx * ny)
# 第三层，调用 add_rational 函数进行运算

# 错误示例，打破抽象屏障
add_rational( [1, 2], [1, 4] )

def divide_rational(x, y):
    return [ x[0] * y[1], x[1] * y[0]]
```

- Abstraction Barrirers
  将有理数看成整个数据、分子和分母；不同层用不同的函数.

- dictionary: 字典具有无序性，是键值对，不能把列表作为键

```
# 字典
numerals = {'I': 1, 'V': 5, 'X': 10}
>>> numerals['X']
10
>>> 'I' in numerals
True
>>> 1 in numerals.values()
True
>>> 'I' in numerals.keys()
True

>>> {x: x*x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> {[1] : 2}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

函数式编程的一个很大的好处就是它可以把**不同**的函数中**相似**的过程抽象出来进行**封装**。例如：

假设 `term` 是一个函数，接收 `n` ，返回 `term(n)` ，例如可以输入 `term = lambda x : x * x`

```python
# 累乘函数
def product(n, term):
    """
    >>> product(3, lambda x : x)
    6
    >>> product(2, lambda x : x**2)
    5
    """
    product, k = 1, 1
    while k <= n:
        product, k = product * term(k), k + 1
    return product

# 累加函数
def summation(n, term):
    total, k = 0, 1
    while total <= n:
        total, k = total + term(k), k + 1
    return total
```

注意到 `product` 和 `summation` 结构是相似的. 稍加封装:

```python
def accumulate(combiner, base, n, term):
    total, k = base, 1
    while k <= n:
        total, k = combiner(total, term(k)), k + 1
    return total

def product(n, term):
    accumulate(add, 0, n, term)

def summation(n, term):
    accumulate(mul, 1, n, term)
```

当然对于上述功能，我们也可以用递归来实现.

```
def summation(n, term):
    """Return the sum of the first n terms in the sequence defined by term. Implement using recu

    >>> summation(5, lambda x: x * x * x) # 1^3 + 2^3 + 3^3 + 4^3 + 5^3
    225
    >>> summation(9, lambda x: x + 1) # 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
    54
    >>> summation(5, lambda x: 2**x) # 2^1 + 2^2 + 2^3 + 2^4 + 2^5
    62
    # Do not use while/for loops!
    """
    if n == 1:
        return term(n)
    return term(n) + summation(n-1, term)
```

- lambda 表达式：作为匿名函数，方便了函数的定义.
- 高阶函数：接受函数作为参数 **或** 返回值是函数的函数. 例子如下：

```
def sum(n, term):
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

def sum_cubes(n):
    return sum(n, lambda x: x**3)
```

Lec 15. Mutable Values

```
from datetime import date
today = date(2015, 2, 20)

"""
>>> today.year
2015
>>> today.strftime('%A %B %d')
'Friday February 20'
"""
```

Lec 17. Iterations

列表（list） 和 迭代器（iterator） 是两种不同的数据结构.

```python
# 将 iter1, iter2, ... 的元素按索引匹配，返回 zip 迭代器(需要 list())。
names = ["Alice", "Bob", "Charlie"]
scores = [85, 90, 95]
combined = list(zip(names, scores))
print(combined)  # [('Alice', 85), ('Bob', 90), ('Charlie', 95)]
```

Lec 18. Objects

- Class: combines and abstracts data and functions 类似于草图
- Object: instantiation of a class 类似于具体建筑
    - String 是内置的 class, append 是函数
    - Int 是内置的 class, + 是函数

      `__init__` 方法
- Attribute: object 的 attribute 是与 object 相关的 name-value pair(名称-值对)
    - Instance attribute: 特定于某个对象的属性.

```
# 银行账户存取款
class Account:
    interest = 0.02
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

tom_account = Account('Tom') # instance
jim_account = Account('Jim')
tom_account.deposit(100)
>>> tom_account.holder
'Tom'
>>> getattr(tom_account, 'balance')
0

>>> tom_account.interest
0.02
Account.interest = 0.04
>>> tom_account.interest
0.04
tom_account.interest = 0.08 # 不改变Class中的interest值
>>> jim_account.interest
0.04
```

Lec 19. Inheritance

```
# Subclass, 提现支付 1 元
class CheckingAccount(Account):
    "A bank account changes for withdrawls."
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount): # 覆盖base class's attribute
        return Account.withdraw(self, amount + self.withdraw_fee)
        # return Account.withdraw(self, amount + CheckingAccount.withdraw_fee) 无法覆盖值.

# Composition
class Bank:
    """
    >>> bank = Bank()
    >>> john = bank.open_account('John', 10)
    >>> jack = bank.open_account('Jack', 5, CheckingAccount)
    >>> john.interest
    0.02
    >>> jack.interest
    0.01
    >>> bank.pay_interest()
    >>> john.balance
    10.2
    """
    def __init__(self):
        self.accounts = []

    # kind = Account 可更改
    def open_account(self, holder, amount, kind = Account):
        account = kind(holer)
        account.deposit(amount)
        self.accounts.append(account)
        return account

    def pay_interest(self):
        for a in self.accounts:
            a.deposit(a.interest * a.amount)

    def too_big_to_fail(self):
        return len(self.accounts) > 1
```

```python
# 最抽象的一集
class A:
    z = -1
    def f(self, x):
        return B(x-1)

class B(A):
    n = 4
    def __init__(self, y):
        if y:
            self.z = self.f(y)
        else:
            self.z = C(y+1)

class C(B):
    def f(self, x):
        return x

"""
a = A()
b = B(1)
b.n = 5
>>> C(2).n

>>> a.z == C.z

Which evaluates to an integer?
b.z # = B(0)
b.z.z # = C(1)
b.z.z.z # = 1
b.z.z.z.z
"""

class SavingAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

# Warning: 最好不用multiple inheritance
class AsSeenOnTVAccount(CheckingAccount, Savingaccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1
```

值得注意的是，如果子类没有 **init**(),它会继承类的 **init**()

CS61A 作业

Lab 06: OOP

```
Q1: Bank Account
# 这段不知道用 append, 不知道加形式参数 self.transaction_count
self.transactions.append(Transaction(self.transaction_count, self.balance, self.balance + amoun

# .append 与 += 不同
lst = [1, 2, 3]
lst.append([4, 5])
print(lst)  # 输出: [1, 2, 3, [4, 5]]  （嵌套列表）
len(lst)  # 输出: 4

lst = [1, 2, 3]
lst += [4, 5]
print(lst)  # [1, 2, 3, 4, 5], 起连接作用.
```

笨办法学python

ex7

```
input print(end1 + end2, end=' ')
print(end3)
output we a
# 以空格结尾，没有换行
```

ex16

```
from sys import argv # argv = ['']

target = open(filename, 'w') # into write mode
print("Something")
target.truncate()
target.write(line1)
target.close() # don't forget it.
```

ex25

```
numbers = [10, 5, 20, 3, 8]
sorted_numbers = sorted(numbers)
print(sorted_numbers)
```

输出结果为：

```
[3, 5, 8, 10, 20]
```

如果输入是英文，则输出按首字母大小排序

```
def sort_words(words):
    return sorted(words) # sorted是内置函数
```

注意，**sorted()** 其实是内置函数，类似于 **print(), range()**

转义字符	功能
\t	ASCII水平制表符(8格)
\n	ASCII换行符
\r	ASCII回车符(回到开头)
Alt+Z	取消换行
Alt+单击	批量选中
Ctrl+Alt+上下	选中多行输入
Ctrl+d	批量选中局部匹配项
Ctrl+u	取消选中局部匹配项
cat xxx	读取文件内容
Ctrl+\	分屏

ex19

```
def rabbit(baby)
    功能……
a = input(">") # 如输入1，则a = "1"
```

lm	$$
\t	
.format('xxx')	

计算机速成课

计算机早期历史

算盘 --> 差分机 --> 继电器 --> 真空管 --> 晶体管(半导体,开关10000次，固体，体积小，小于50nm)
从机电时代进入电子时代，计算速度不断提高(电子计算机需要)

布尔代数

- 中央处理器 (CPU) 负责执行程序，分为取指令 -> 解码 -> 执行.

图片说明

13.算法入门

28.计算机网络