## python3基础知识回顾

| | | | |
|---|---|---|---|
| **笔记本:** | python3基础知识回顾 | | |
| **创建时间:** | 2018/8/8 14:43 | **更新时间:** | 2018/8/9 20:21 |
| **作者:** | 王 | | |
| **URL:** | https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/00143165... | | |

python3两个整数相除结果为浮点数，//除为floor除，结果为精确除

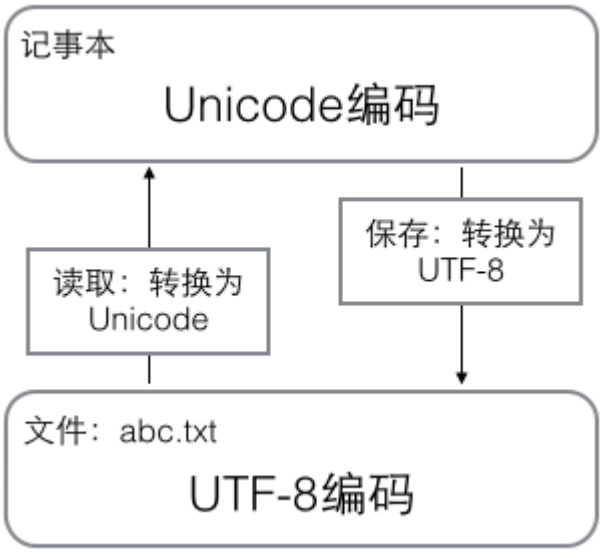
变量的命名必须是大小写英文、数字和_的组合，且不能以数字开头

变量类型不固定的语言称之为动态语言，与之对应的是静态语言，静态语言在定义的时候必须指定数据类型。

a="ABC"
分为两步来看
1.在内存中创建'ABC'字符串
2.在内存中创建a变量，并把它指向'ABC'

python中的整数数据没有大小限制，浮点型数据也没有大小限制，但是超出一定范围之后就会表示为inf

ASCII编码只有大小写英文字母、数字和一些符号，unicode编码虽然解决了乱码问题，并成功表示了所有字符，但是不符合编码最优性原理，所以才设计了utf-8的编码方式。
utf-8把一个unicode字符根据不同的数字大小编写为1-6个字节长度，常用英文1个字节长度，中文通常是3个字节长度。
用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件




python3中，字符串以unicode进行编码
ord()获取字符的整数表示，chr()将编码转换为对应的字符

#coding=utf-8是为了告诉解释器，按照utf-8编码读取源代码，否则中文可能出现乱码

***格式化输出***
方法一

```
print('Hi, %s, you have $%d.' % ('Michael', 1000000))
```

数据类型：%s表示字符串，%d表示整数，%f表示浮点数，%x表示十六进制整数
如果不确定使用什么类型的数据的话，使用%s是万能的
方法二
采用字符串的format函数

```
'Hello, {0}, 成绩提升了 {1:.1f}%'.format('小明', 17.125)
```

***list***
末尾添加：L.append(3)
指定位置添加：L.insert(3,'something')
末尾删除：L.pop()

***tuple***
tuple是不可变元组，在定义的时候就必须将每一个元素都确定下来，之后不可改变，
tuple如果只有一个元素的话，不能采用a=(2)的形式来定义，使用a=(2,)代替

***input***
函数获取的输入值是str类型的，所以不能直接与其他整数类型的数据进行比较。
（不同类型之间可以进行相等的比较，但是不可以进行大小的比较，比如 'if adf'==123，结果为false，但是 if 'adf'>123就会报错）

***range***
函数返回值可以之间转换为list，比如list(range(5))返回值为[0,1,2,3,4]

***dict***
字典数据类型，d.keys()获取键，d.values()获取值，d.items()获取键值对组成的tuple
dict的键是不可变对象，所以它的键不能为list数据类型

***set***
集合的并：s1|s2
集合的交：s1&s2

***str***
str的replace函数不改变原有的str数据的值，只是在拷贝的数据上进行替换操作

***pass***
pass表示什么都不做，可以作为占位符，比如如果还没有想好怎么写函数代码，就可以先放一个pass

函数返回多个值的时候其实返回的是这多个值组成的一个tuple，函数执行完毕之后如果没有return语句，自动执行的是return None

***函数默认参数***
一个最大的坑:

```
def add_end(L=[]):
    L.append('END')
    return L
add_end()
输出：['END']
add_end()
输出：['END','END']
```

究其原因，默认参数也是一个变量，是在函数定义的时候，默认参数就会在内存中创建出来了，解决方法是让L=None这个不可变对象

***可变参数***
```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```
这样就可以对calc函数传递任意数目参数了，在函数调用的时候自动组装为一个tuple
如果想传递一个list的话，需要在调用函数的时候在参数前面加*号，这样就表示把list的所有参数作为一个可变参数传递进去
>>>nums=[1,2,3]

>>>calc(*nums)

***关键字参数***
可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。
关键字参数能够扩展函数的功能

***命名关键字参数***
如果要限制关键字参数的名字，可以使用命名关键字参数，比如只接受city和job作为关键字参数
```
def person(name, age, *, city, job):    #*号后面表示关键字参数
    print(name, age, city, job)
```
命名关键字参数必须传入参数名，`person('Jack', 24, 'Beijing', 'Engineer')会报错`

***参数组合***
组合顺序：必选参数，默认参数，可变参数，命名关键字参数，关键字参数

***递归***
尾递归优化：在函数返回的时候调用函数自身，并且return语句中不能包含表达式，这样编辑器或者解释器就可以把尾递归做优化，无论调用自身多少次，都只占用一个栈
但是大多数编程语言并没有对尾递归做优化（包括python），所以递归还是可能引起栈溢出

***enumerate***
函数将一个list变为一个索引-元素对，类比于range函数
```
for i in enumerate(['A', 'B', 'C']):
    print(i)
```
```
0 A
1 B
2 C
```

***列表生成器（创建一个列表）***
两层循环：`[m + n for m in 'ABC' for n in 'XYZ']`

***生成器***
由于列表生成器生成的列表受到内存的限制，列表的容量是有限的。基于上述原因，创建了生成器。即在循环过程中不断推演出后面的元素，这种一遍循环一遍计算的机制称为生成器：generator（generator保存的是算法）

***生成器的创建方式：***
（1）将列表生成器的[]替换为()
`g = (x * x for x in range(10))`
`利用next(g)可以获取函数的下一个值，但是一般采用for循环来调用generator`
（2）如果一个函数中包含了yield关键字，那么这个函数就不再是一个普通函数，而是一个generator。generator与函数的执行顺序不一样，函数是顺序执行遇到return语句或者到达最后一条语句就会返回，但是如果变成generator的函数，每次执行next的时候，遇到yield返回，再次执行时从上一次返回的yield语句处继续执行。

```
#斐波那契亚数列
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'

#如果想要捕获返回值的话，需要捕获StopIteration错误，返回值包含在value中
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
```

```
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

***迭代器***

可迭代对象（Iterable）：可以直接作用于for循环的对象称为可迭代对象，例如list, tuple, dict, set, str, generator等
迭代器（Iterator）：不但可以作用于for循环，还可以被next()不断调用返回下一个值
迭代器都可以强制转换为list

可迭代对象和迭代器的判断：

```
from collections import Iterable,Iterator
isinstance(a,Iterable)
isinstance(a,Iterator)
```

生成器都是迭代器
python的for循环本质上是通过不断调用next函数实现的，例如：

```
for x in [1, 2, 3, 4, 5]:
    pass
```

**等价于：**

```
    # 首先获得Iterator对象:
    it = iter([1, 2, 3, 4, 5])
    # 循环
    while True:
        try:
            # 获得下一个值:
            x = next(it)
        except StopIteration:
            # 遇到StopIteration就退出循环
            break
```

***map***

接收两个参数，第一个是函数，第二个是Iterable，map将传入的函数依次作用到Iterable中的每个元素，结果返回一个新的Iterator

***reduce***

reduce接收的第二个参数可以是Iterable

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

reduce把一个函数作用在一个序列（Iterable）上，这个函数必须接收两个参数，然后reduce把得到的结果继续和下一个元素做函数运算。

***filter***

类似于map，filter接收一个函数和一个Iterable对象，返回一个Iterator，和map不同的是，filter作用于每个元素，然后根据返回值是True还是False来确定保留还是丢弃这个元素，True保存，False丢弃
例如删掉一个序列中的空字符串

```
def not_empty(s):
    return s and s.strip()
```

```
list(filter(not_empty, ['A', '', 'B', None, 'C', ' ']))
# 结果: ['A', 'B', 'C']
```

***sorted***

排序函数默认按照从小到大的顺序，字符串的话按照ASCII码进行排序
sorted可以接受一个key函数，排序规则是按照这个key函数的返回值来设定的
反向排序

```
sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)
```

***返回函数***

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
f=lazy_sum(1,2,3)
f()
```

#函数返回的是另一个函数，但是返回的函数并没有立刻被执行，只有在调用函数的时候才会产生结果，相关参数和变量都保存在返回的函数中,这种程序结构称为闭包

一个典型的example

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i*i
    fs.append(f)
    return fs
f1, f2, f3 = count()
f1()    #9
f2()    #9
f3()    #9
```

原因是返回的函数并不是立刻执行的，而是调用的时候才会执行，等到三个函数都返回的时候，他们引用的变量i已经变成了3，所以结果为9

所以使用闭包的时候返回函数不能引用任何循环变量，或者后续会发生变化的量

***装饰器***

在代码运行期间动态添加功能的方式
比如，我们要定义一个能够打印日志的decorator，可以这样写

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper

@log
def now()
    print(time")
```

这样在调用函数now的时候，会先打印now函数的名字，然后执行now函数
在函数前面添加@log，相当于now=log(now)，但是这样的话调用now.__name_会发现now的名字改为了wrapper，解决方法是在wrapper函数定义的上面写上@functools.wraps(func)

***偏函数***

当函数的参数很多的时候，可以利用偏函数创建一个新的函数，这个新的函数可以固定住（设置默认值）原函数的部分参数，从而在调用的时候更加简单

```
int2 = functools.partial(int, base=2)
```