函数(二)

- 在函数(一)中，我们讨论了函数的定义、函数的四种参数、以及调用函数的方法。你已经完整学会了一套使用函数的范式。

当你日后使用函数时，60%的情况可以被函数(一)囊括，剩余30%的情况可以被后续的面向对象覆盖，只有10%的情况需要用到今天的内容

- 这节课将从一个更高的视角来讨论函数，我们会将函数这一概念拓展到函数对象，进而像使用任何变量一样灵活地使用函数。

函数对象与函数名

- 在函数(一)中，我们通过def来定义函数，或者说创建了一个**函数对象(在类型标注时写为callable)**。用def创建的函数对象，可以认为是一种**常量**
- 函数名是函数对象的一部分，我们可以通过访问函数对象的**name**变量来获取函数名
- 用函数名调用函数，类似于直接使用一个常量

```
if cond1:

func1()

else:

func2()



# 类似于

print([1,2,3])
```

函数数据化

- 在Python中，函数可以认为是一种广义的数据。我们也可以用一个变量来存储函数，再用这个变量来进行函数调用，类似于使用变量。

需要特别注意的是，以下代码中的f是一个**变量名**，它不属于函数对象的一部分

```
'''
这句的含义是将函数func1或func2存储到变量f上，
不需要在函数名后加括号，
否则会进行函数调用，得到的是函数的返回值。
'''


f = func1 if cond1 else func2
f()                                 # 调用变量f中存储的函数


# 类似于
l = [1, 2, 3]
print(l)
```

- 既然函数也是数据的一种，也可以被变量存储与传递，我们可以自然地想到，函数本身也可以作为函数参数与返回值。我们先介绍函数作参数的情况：

```
# 函数作函数参数


# 先定义需要用到的函数
def mean(l):
```

```
...

def majority(l):

...

def median(l):

...



# process参数接受一个函数

def agg(l: list, process: callable):

        # 统一处理l的输入是否合法

        some code

        return process(l) # 统一处理
```

下面是Python的一些内置函数用到函数作参数的实例

```
l = [1, 2, 2, 3]



# 这里的l.count就是一个函数，key可以接受任意接口吻合的函数

maximum = max(l, key = l.count)

l.sort(key = l.count)
```

map、reduce和filter也是相当典型的例子

```
from functools import reduce
```

```
def add(x, y):
        return x + y
def pow(x):
        return x ** 2
def is_odd(x):
        return x % 2 == 1


l = [1, 2, 3]
l1 = filter(is_odd, l)      # 用check筛选l中的值，得到筛选后的列表的迭代器
l1 = map(pow, l1)           # 把一元函数pow作用于l中的每个值，得到计算后的新列额表
res = reduce(add, l1)       # 用二元函数add聚合l中的所有元素，得到结果
```

匿名函数

至此，我们发现通过参数来传递函数并非是一种炫技，而是一种常用的做法，这可以帮助我们实现函数的拆分与组合，提高灵活性。

但我们很快想到一个新的问题，有时候我们只需要表达一个很简单的逻辑，比如前面的add和pow，它们的实现都只需要一行代码，有时为了能进行传递却必须特意到文件外面去定义函数，这加重了我们的负担。因此，对于那些逻辑极为简单的函数，Python提供了一种新的定义方式：匿名函数

```
# 格式: lambda arg1, arg2...: return_value
```

```
l = [1, 2, 3]

l1 = filter(lambda x: x % 2 == 1, l)

l1 = map(lambda x: x ** 2, l1)

sum = reduce(lambda x, y: x + y, l)
```

以这种方式被创建的函数对象没有特定的函数名，因为它并不期待被复用。但如果你想在小范围内对其进行复用，可以用一个变量来存储它

```
f = lambda x: x > 0         # f是变量名而不是函数名

print(f(1), f(0))
```

匿名函数的另一个常见应用场景是用来组合函数

```
def composite(f, g):

        return lambda x: f(g(x))    # 想想为什么不能直接return f(g(x))



# 如果你把lambda用上头了的话

composite = lambda f, g: (lambda x: f(g(x)))



h = composite(f, g)             # 实现了数学上的h(x) := f(g(x))
```

匿名函数也非常适合用来从多变量函数中获取偏函数

```
pow = lambda x, n : x ** n
```

```
pow_2 = lambda x: pow(x, 2)

pow_3 = lambda x: pow(x, 3)

print(pow_2(2), pow_3(2))
```

闭包(了解即可)

在使用匿名函数的过程中，我们发现将函数作为返回值也是十分常见的情况。我们经常需要定义一些用于生产函数的函数，这是因为我们希望将函数的**参数配置过程与函数的执行过程分离，达到一次配置、多次调用的效果**。

```
# 定义了生产一类范围过滤函数的函数



def range_filter(range_left, range_right):

        def filt(l):

                return [x for x in l if x >= range_left and x <
range_right]


        return filt



my_filter = range_filter(1,5) # 实例化了一个具体的过滤函数

l = [1, 5, 8, 12]

l = my_filter(l) # 调用具体的过滤函数
```

- 从以上代码可以观察到，作为返回值的函数的定义依赖于函数range_filter的内部变量range_left、range_right，这些变量本应在range_filter函数退出后就被丢弃了，但他们的值却被my_filter继续使用，我们称my_filter捕获了它的外部变量。这样的结构被称为闭包。

- 注意点1:捕获的外部变量必须是**不可变对象**,否则生成的函数的功能将是不稳定的
- 注意点2:当你试图在闭包里修改捕获的外部变量时，请用**nonlocal修饰变量名**来帮助解释器明白你并不是想创造一个新的内部变量

CS61A中讲到了如何用复杂的闭包来模拟一个类来帮助我们加深对类和闭包的理解，但我们的目的只是快速学会写出可用而清晰的代码。所以，更多时候，我们会直接使用类来实现闭包的功能而不是反过来。

装饰器(了解即可)

装饰器是一个非常有python特色的功能。它的作用是为已有的函数增添一些功能，而不必对函数做侵入式的修改。装饰器在fastapi/flask等后端框架中很常见，但我们自己写代码时用到装饰器的场景还是比较少的，所以这块内容以了解为主。

```
def log(fn):

        def wrapper(*args, **kw):

                print(f'call function: {fn.__name__}()')

                return fn(*args, **kw)

        return wrapper



# 这里的@log 等价于add = log(add)



@log

def add(x, y):

        return x + y



add(1, 2)
```

装饰器也支持带参数的形式，只需要编写一个返回一个装饰器函数的函数即可

递归(了解即可)

有时候，我们希望以一种**从后向前的视角**来解决问题，即我们总是只考虑如何做好事情的最后一步，这种视角对应的程序实现就是使用递归

用递归实现阶乘

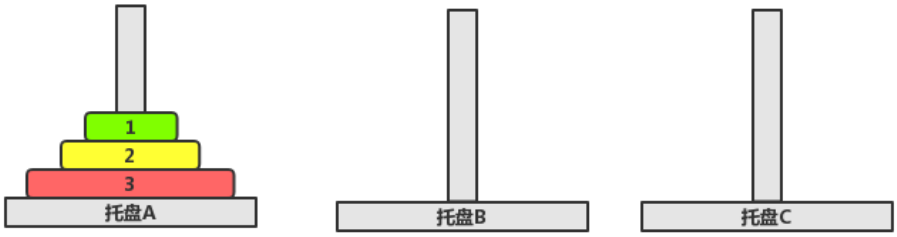
```
def fact(N):
        # 考虑递归的结束条件
        if N == 1:
        return 1

        # 具体递归过程
        else:
                return N * fact(N - 1)
```

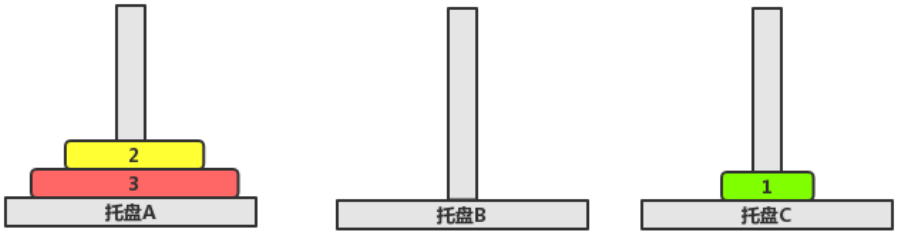
经典中的经典:汉诺塔问题

汉诺塔算法图解

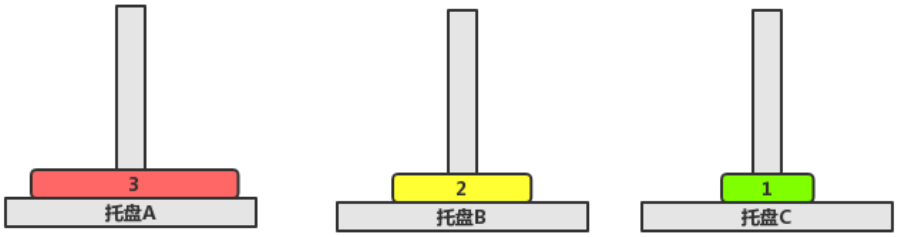
准备：以三个盘子（1、2、3）为例，借助托盘B从托盘A上的盘子移动至托盘C



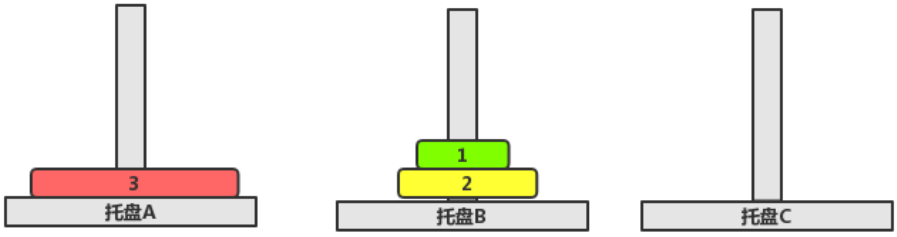
第一步：将盘子1从托盘A移动至托盘C



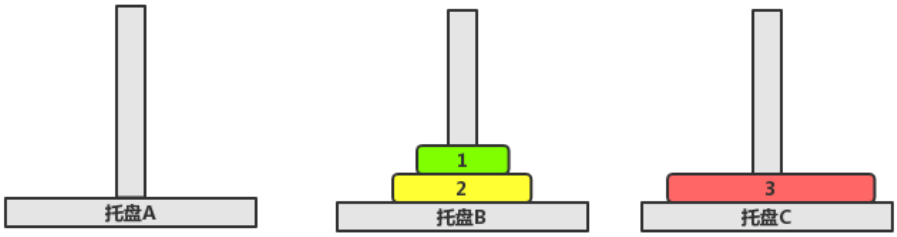
第二步：将盘子2从托盘A移动至托盘B



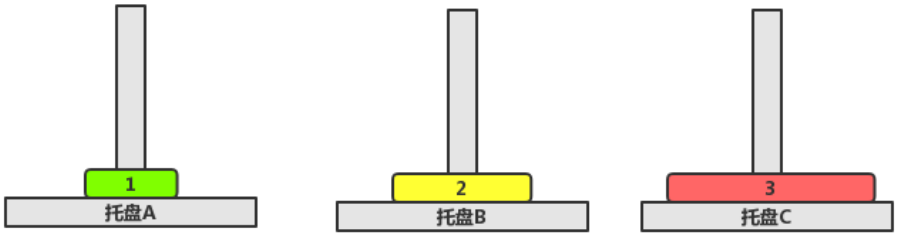
第三步：将盘子1从托盘C移动至托盘B



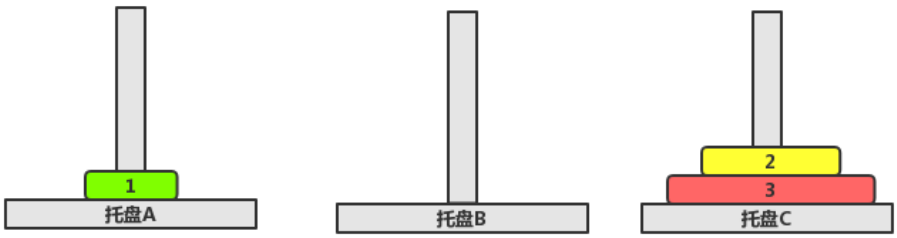
第四步：将盘子3从托盘A移动至托盘C



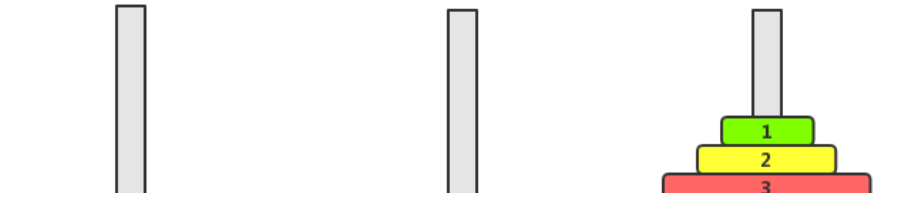
第五步：将盘子1从托盘B移动至托盘A



第六步：将盘子2从托盘B移动至托盘C



第七步：将盘子1从托盘A移动至托盘C



```
def move(n, A, B, C):
        if n==1:
                print(f'Move {n} from {A} to {C}')
        else:
                move(n-1, A, C, B)
                move(1, A, B, C)
                move(n-1, B, A, C)

move(5, 'A','B','C')
```

在算法题中，递归是非常常见的思考方向，尤其是在树结构相关的题目中。当然，递归往往并不是一种高效的实现，因为总是需要保留多个函数的栈帧，不经特殊优化(如尾递归)很容易出现**栈空间溢出**的情况，故而并不常用。

yield与生成器

我们讲for循环时曾提到过，range函数并没有一次性计算出所有满足条件的值，而是每次调用时再去计算，这是一种**惰性策略**。类似地，map、filter方法返回的也只是一个**迭代器**，只有当你试图遍历map对象或filter对象时，它们才会去具体计算需要返回的东西。这种惰性策略的好处是，不用一次性加载过多的数据在内存中，避免了还没用到甚至永远用不到的数据的计算浪费与存储浪费。

我们有时希望我们自己定义的函数也有这样的功能，只在真正被需要时才计算部分数据，那么我们可以用yield来替代return以实现这一功能。yield与return的不同之处在于执行yield时不会退出函数，而会暂停并等待下次计算请求。

```
# 生成fib序列的前N项的生成器
```

```
def fib(N):

        a, b = 0, 1

        for _ in range(N + 1):

                a, b = b, b + a

                yield a



fib_nums = fib(10)

for x in fib_nums:

        print(x)
```

在实际应用场景中，如果一个函数本应设计为返回一个列表，且这个列表后续将仅被for循环使用且往往只会用到前几项，那么就可以考虑把函数设计为用yield返回的形式。

举个例子，在爬虫框架scrapy中，你会频繁使用yield，因为你的代码逻辑应该是爬一点处理一点，而不是爬取一大堆东西塞满你的内存再开始进行处理。

当然，生成器这一概念并不与yield绑定。在我们需要创建一个特别大的列表供循环使用时，也可以考虑将其改为生成器，这只需要我们在列表推导式上做微小的改动:把[]改为()即可

```
l = [x ** 2 for x in range(10000000)] # 这是一个超大列表

g = (x ** 2 for x in range(10000000)) # 这是一个生成器



# 像遍历列表一样遍历生成器



for x in g:

        if x % 100000 == 0:
```

```
        print(x)
```