# CS61A NOTE9 String Representation,Efficiency

## Representation: Repr,Str

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes.Python 有两种方法产生一个对象的 "字符串"：str()和 repr()。两者相似，但用途不同。

`str()` is used to describe the object to the end user in a "Human-readable" form, while `repr()` can be thought of as a "Computer-readable" form mainly used for debugging and development. str()以 "人类可读 "的形式向终端用户描述对象，而 repr()是 "计算机可读 "的形式，用于调试和开发。

When we define a class in Python, `__str__` and `__repr__` are both built-in methods for the class. 定义类时__str__和__repr__都是该类的内置方法。

We can call those methods using the global built-in functions `str(obj)` or `repr(obj)` instead of dot notation, `obj.__repr__()` or `obj.__str__()`.可用全局内置函数 str(obj)或 repr(obj)调用，而非用点符号 obj.**repr**()或 obj.**str**()。

In addition, the `print()` function calls the `__str__` method of the object and displays the returned string **with the quotations removed**, while simply calling the object in interactive mode in the interpreter calls the `__repr__` method and displays the returned string **with the quotations removed**.打印 print()函数调用对象的__str__方法，显示 return 字符串并去掉引号。在解释器中以交互式模式调用对象调用_repr__方法，显示 return 字符串并去掉引号

打印：总触发__str__,单有__repr__才触发__repr__ ， 因此 str 可读性强但不准确且 str 影响 repr

直接输出：有__repr__才触发__repr__，否则不变

纯文本

```
class Rational:#有理数

    def __init__(self, numerator, denominator):#分子、分母
        self.numerator = numerator
        self.denominator = denominator #一会儿外套{}

    def __str__(self): #
        return f'{self.numerator}/{self.denominator}' #写法f'...'

    def __repr__(self):
        return f'Rational({self.numerator},{self.denominator})'

>>> a = Rational(1, 2)
>>> str(a) #有引号
'1/2'
>>> repr(a) #有引号
'Rational(1,2)'
>>> print(a) #打印去除引号
```

```
1/2
>>> a
Rational(1,2) #本身不变
```

纯文本

```
>>> s = "hello" # Python String objects also have __repr__ and __str__ meth

>>> repr(s)  #直接输出：有__repr__才触发__repr__，否则不变
"'hello'"

>>> s 直接输出：无__repr__，不变
'hello' # displays the repr string with the outer layer of quotations remov

>>> print(repr(s)) 打印：单有__repr__触发__repr__，但print无外面引号
'hello' # printing the repr string removes the outer layer of quotations  夕

>>> str(s) 直接输出：无__repr__，不变
'hello'

>>> print(s) 打印：都无不变s，打印去引号
hello
```

打印：总触发__str__,单有__repr__才触发__repr__ ， 因此 str 可读性强但不准确且 str 影响 repr

直接输出：有__repr__才触发__repr__，否则不变

## Q5:WWPD Repr-esentation

纯文本

```
class A:
    def __init__(self, x):
```

```
        self.x = x

    def __repr__(self):
         return self.x

    def __str__(self):
         return self.x * 2

class B:
    def __init__(self):
         print('boo!')
         self.a = []

    def add_a(self, a):
         self.a.append(a)

    def __repr__(self):
         print(len(self.a))
         ret = ''
         for a in self.a:
             ret += str(a)
         return ret

>>> A('one')
one   #直接输出有__repr__触发__repr__，为什么没有''，直接输出要去''
>>> print(A('one')) #打印总触发__str__，print去掉''
oneone
>>> repr(A('two'))
'two'  #repr或str仍有''

>>> b = B() #直接输出虽然有repr但没输入没啥可打印/return的，因此只poo！
boo!   #打印去引号
>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b  #直接输出有__repr__触发__repr__
2
aabb #str是2项，为什么没有'',直接输出要去''

直接输出：有__repr__才触发__repr__，A('a')是字符串a，A('b')是字符串b，
self.a得到[a,b],打印长度2
对i=a,str(a)等价于a.__str__,得到字符串aa，对i=b,str(b)等价于b.__str__,得到字符串b
#str(obj)或repr(obj)调用，而非用点符号obj.repr()或obj.str()
```

## Q6: Cat Representation

Now let's implement the `__str__` and `__repr__` methods for the `Cat` class from earlier so that they exhibit the following behavior:现在让我们为前面的猫类实现__str__和__repr__方法，使它们表现出以下行为：

纯文本

```
>>> cat = Cat("Felix", "Kevin")
>>> cat
Felix, 9 lives 直接输出：猫名，lives
>>> cat.lose_life()
>>> cat
Felix, 8 lives 直接输出：猫名，lives-1
>>> print(cat)
Felix 打印：猫名
既有__repr__也有__str__，打印只__str__猫名，直接输出：有__repr__触发__repr__猫名，
```

打印：总触发__str__,单有__repr__才触发__repr__ ， 因此 str 可读性强但不准确且 str 影响 repr

直接输出：有__repr__才触发__repr__，否则不变

直接输出或 print 要去掉''

str/repr 保留''

纯文本

```
# (The rest of the Cat class is omitted省略 here, but assume all methods fro

    def __repr__(self): #猫名，lives。用f'{}'
        "*** YOUR CODE HERE ***"
        return f'{self.name}, {str(self.lives)} lives'
        #return f'{self.name}, {self.lives} lives'
        #我最爱😍 注意前面只输出数字，后面还有lives
        #return self.name + ", "+str(self.lives)

    def __str__(self): #猫名。用f'{}'
        "*** YOUR CODE HERE ***"
        return f'{self.name}'
```

**Example**

```
class Dinosaur:
    def __init__(self, weight):
        self.weight = weight
    def __str__(self):
        if self.weight < 50:
            return “squeak!”
        elif self.weight < 100:
            return “rawr!”
        else:
            Return “ROAR!”
    def __repr__(self):
        return “Dinosaur(“ + str(self.weight) + “)”
```

```
>>> dave = Dinosaur(40)
>>> print(dave)
squeak

>>> dave.__str__()
“squeak”

>>> dave.__repr__()
“Dinosaur (40)”

>>> dave
Dinosaur (40)
```

'sth' + str(变化的东西) + 'sth'

# Efficiency

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by *runtime*?函数的效率时:随着输入规模增长，函数的运行时间如何变化?

**Example 1:** `square(1)` 1*1 requires one primitive operation: multiplication. `square(100)` 100*100 also requires one. No matter what input `n` we pass into `square`, it always takes a *constant* number of operations (1) In other words, this function has a runtime complexity of Θ(1).运行时间/操作数是常数，运行时间复杂性为 Θ(1)

**Example 2:** `factorial(1)` 阶乘 requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases **linearly** proportional to the input. In other words, this function has a runtime complexity of Θ( `n` ).运行时间（操作数）与输入成线性比例增加，运行时间复杂性为 Θ(n)。

**Example 3:**

纯文本

```
def bar(n):
    for a in range(n):
        for b in range(n):
            print(a,b)
```

`bar(1)` requires 1 print statements, while `bar(100)` requires `100*100 = 10000` print statements (each time `a` increments, we have 100 print statements due to the inner for loop). Thus, the runtime increases **quadratically** proportional to the input. In other words, this function has a runtime complexity of Θ( `n^2` ).两次循环，函数的运行时间复杂性为 Θ(n^2)

**Example 4:**

纯文本

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

`rec(1)` requires one addition, as it returns `rec(0) + rec(0)`, and `rec(0)` hits the base case and requires no further additions. but `rec(4)` requires `2^4 - 1` = 15 additions. To further understand the intuition, we can take a look at the recurisve tree below. To get `rec(4)`, we need one addition. We have two calls to `rec(3)`, which each require one addition, so this level needs two additions. Then we have four calls to `rec(2)`, so this level requires four additions, and so on down the tree. In total, this adds up to 1 + 2 + 4 + 8 = 15 additions.对 n 有 2**n–1 次操作。n 增加，指数(2)增加

当我们增加 n 的输入大小时，运行时间（操作数）与输入成指数比例地增加。换句话说，这个函数的运行时间复杂性为 Θ(2^n)。

Here are some general guidelines for finding the order of growth for the runtime of a function:函数运行时间规律

- If the function is recursive or iterative, you can subdivide the problem as seen above:递归或迭代
  > - Count the number of recursive calls/iterations that will be made in terms of input size `n`.递归调用步骤
  > - Find how much work is done per recursive call or iteration in terms of input size `n`.每次递归或迭代要做多少步
  > - The answer is usually the product 乘积 of the above two, but be sure to pay attention to control flow!注意控制流
- If the function calls helper functions that are not constant-time, you need to take the runtime of the helper functions into consideration.如果调用的辅助函数不是固定值则需要考虑辅助函数
- We can ignore constant factors. For example `1000000n` and `n` steps are both linear.忽略常数因子

- We can also ignore smaller factors. For example if `h` calls `f` and `g`, and `f` is Quadratic 二次函数 while `g` is linear 线性函数, then `h` is Quadratic.忽略更小的因子
- For the purposes of this class, we take a fairly coarse 粗糙 view of efficiency. All the problems we cover in this course can be grouped as one of the following:
    - Constant: the amount of time does not change based on the input size. Rule: `n --> 2n` means `t --> t`.常数
    - Logarithmic: the amount of time changes based on the logarithm of the input size. Rule: `n --> 2n` means `t --> t + k`.对数
    - Linear: the amount of time changes with direct proportion to the size of the input. Rule: `n --> 2n` means `t --> 2t`.线性
    - Quadratic 次方: the amount of time changes based on the square of the input size. Rule: `n --> 2n` means `t --> 4t`.二次
    - Exponential: the amount of time changes with a power of the input size. Rule: `n --> n + 1` means `t --> 2t`.不是 e 就是指数分布

## disc Q6: The First Order...of Growth

纯文本

```
What is the efficiency of rey?

def rey(finn):
    poe = 0
    while finn >= 2:
        poe += finn
        finn = finn / 2
    return
    n*（2-1/2**（n-1））

    #n --> 2n means t --> t + 1因此是logarithmic对数O(log(finn))
```

纯文本

```
def mod_7(n):
```

```
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
    #⚠️ 易错，不是线性n，而是常数字O(1)，不过0-7次罢了不随n增长稳定增长
```

Constant, since in the worst case scenario our function `mod_7` will require 6 recursive calls to reach the base case. Consider the worst case where we have an input `n` such that our first call to `mod_7` evaluates `n % 7` as `6`. Each recursive call will decrement `n` by `1`, allowing us to eventually reach the base case of returning `0` in 6 recursive calls (`n` will range from 0 to 6). Since the growth of the computation is independent of the input, we say this is constant, which is commonly known as a Θ(`1`) runtime.常数，因为在最坏的情况下，我们的函数 mod_7 将需要 6 次递归调用才能达到基本情况。由于计算的增长与输入无关，我们说这是恒定的，这就是通常所说的 Θ(1)运行时间。

## lab9 WWPD

纯文本

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

Linear θ(n)
In the worst case, n is prime, and we have to execute the loop n - 2 times.
```

纯文本

```
def bar(n):
    i, sum = 1, 0
    while i <= n:
        sum += biz(n)
        i += 1
    return sum

def biz(n):
    i, sum = 1, 0
    while i <= n:
        sum += i**3
```

```
        i += 1
    return sum

Quadratic Θ(n^2)
#n*n
```

纯文本

```
def foo(lst, i):
    mid = len(lst) // 2
    if mid == 0:
        return lst
    elif i > 0:
        return foo(lst[mid:], -1)
    else:
        return foo(lst[:mid], 1)

Logarithmic Θ(log(n))
#n --> 2n means t --> t + 1
```

## lab10 WWPD

纯文本

```
def count_partitions(n, m):
    """Counts the number of partitions of a positive integer n,
    using parts up to size m."""
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
Exponential,画图二叉树

def is_palindrome(s):
    """Return whether a list of numbers s is a palindrome."""
    return all([s[i] == s[len(s) - i - 1] for i in range(len(s))])
Linear
```

```
def binary_search(lst, n):
    """Takes in a sorted list lst and returns the index where integer n
    is contained in lst. Returns -1 if n does not exist in lst."""
    low = 0
    high = len(lst)
    while low <= high:
        middle = (low + high) // 2
        if lst[middle] == n:
            return middle
        elif n < lst[middle]:
            high = middle - 1
        else:
            low = middle + 1
    return -1
Logarithmic
```

exam practice

**6. (1.0 points) Naming Is Hard**

Consider this function that Pamela wrote to brainstorm baby names:

```
def name_options(names):
    """
    Returns possible combinations of first and middle names based on NAMES list,
    where a first name and middle name should not be the same.
    Any of the names in NAMES can be either a first or a middle name.

    >>> name_options(['Sequoia', 'Alexia'])
    ['Sequoia Alexia', 'Alexia Sequoia']
    >>> name_options(['Sierra', 'Elantris', 'Maritima', 'Armeria'])
    ['Sierra Elantris', 'Sierra Maritima', 'Sierra Armeria',
     'Elantris Sierra', 'Elantris Maritima', 'Elantris Armeria',
     'Maritima Sierra', 'Maritima Elantris', 'Maritima Armeria',
     'Armeria Sierra', 'Armeria Elantris', 'Armeria Maritima']
    """
    return [ f'{first} {middle}' for first in names
             for middle in names if first != middle]
```

**(a) (1.0 pt)** What is the order of growth of `name_options` in respect to the size of the input list `names`?

- ○ Constant
- ○ Logarithmic
- ○ Linear
- ● Quadratic
- ○ Exponential