Python 101

Lec03
Functions

thoum

August 12, 2019

Functions

$$f(x) = x + 1$$
$$f(x) = \ln x$$

Python Functions

```python
def f(x):
    # code block belonging to the function
    r = x+1
    return r
# end of function

def g(x):
    return math.log(x)

# Note that each x is unique to the function

# Calling the function
print(f(3)) # f(3) == 4
print(f(3))
```

Why Functions?

- Reusability
- Abstraction
- And many more

Reusability

```
a = [[1,2],[3,4]]
b = [[3,2],[1,3]]
c = [[1,4],[3,5]]
d = [[3,2],[8,1]]

res = [[0 for x in range(len(a))] for x in
    range(len(b))]
for k in range(len(b[0])):
    for i in range(len(a)):
        r = a[i][k]
        for j in range(len(b)):
            res[i][j] += (r * b[k][j])
res2 = [[0 for x in range(len(c))] for x in
    range(len(d))]
for k in range(len(d[0])):
    for i in range(len(d)):
        r = c[i][k]
        for j in range(len(d)):
            res[i][j] += (r * d[k][j])
# and once more
```

Reusability

```
def mat_mul(m0, m1):
    """crude matrix multiplicaiton"""
    res = [[0 for x in range(len(m0))] for x in
        range(len(m1))]
    for k in range(len(m1[0])):
        for i in range(len(m0)):
            r = m0[i][k]
            for j in range(len(m1)):
                res[i][j] += (r * m1[k][j])
    return res

a = [[1,2],[3,4]]
b = [[3,2],[1,3]]
c = [[1,4],[3,5]]
d = [[3,2],[8,1]]

t = mat_mul(a,b)
k = mat_mul(c,d)
mat_mul(t, k)
```

Reusability

- Less code to read
- Fix once, fix everywhere
- DRY: Don't Repeat Yourself

Abstraction

```
print()
```

vs

```
builtin_print ( PyObject * self , PyObject * args ,
PyObject * kwds )
{
  static char * kwlist [] = {" sep ", " end ", " file
     ", 0};
  static PyObject * dummy_args = NULL ;
  static PyObject * unicode_newline = NULL ,
    * unicode_space = NULL ;
  static PyObject * str_newline = NULL ,
    * str_space = NULL ;
  PyObject * newline , * space ;
  PyObject * sep = NULL , * end = NULL , * file = NULL ;
  int i , err , use_unicode = 0 ;
  /*and much more*/
```

Abstraction

- Lets us focus

Understanding Functions: Pure

```python
def my_pow(x, y):
    return x**y

# giving clear names to functions is important
def factorial(n):
    ret = 1
    for i in range(1, n+1):
        ret *= i
    return ret

def quadruple(x):
    return 4 * x
```

Understanding Functions: Impure

```
x = [1,2,3]
def change_io():
    print("Input/Output state changed")

def change_element_of_list(l):
    l[0] = 3
change_element_of_list(x)
print(x)

x = 3
def change_4(y):
    y = 4
change_4(x)
print('x is now:', x)

def change(x):
    print('x:', x)
    x = [4,5,6]
    print("x changed:", x)
change(x)
print('x is now:', x)
```

Understanding Functions: Impure

```python
y = [5, 6, 7]
def change_by_global():
    global y
    print('y:', y)
    y = [8,9,10]

print("y changed:", y)
print('y is now:', y)

change_by_global()
print('y is now:', y)
```

Detour: Understanding Binding

Function parameters and variables can be thought as labels.
The variables names(labels) are *local* to the function, called the *scope* of the variable.

Detour: Understanding Binding

Here, the local l points to the same list x is pointing.
By $l[0] = 3$, that list(the one x, y is pointing)'s first element points to 3.
Thus, it changes x.

```
x = [1,2,3]
def change_element_of_list(l):
    l[0] = 3
change_element_of_list(x)
print(x)
```

Detour: Understanding Binding

Here, the local y points to the same 3 x is pointing.
By $y = 4$, the local y now points to 4. (x, y nows points to different things)
Thus, $x == 3$.

```
x = 3
def change_4(y):
    print(y)
    y = 4
    print(y)
change_4(x)
print('x is now:', x)
```

Detour: Understanding Binding

We use *global* to *assign* value to a name defined outside the function. We can use the values without using *global*, but not recommended in large programs as it is unclear where the variable was defined.

```
y = [1,2,3]
def change_by_global():
    global y
    y = 3
print(y)
```

Detour: Understanding Binding

Here, the global y points to the y outside the function.
By $y = 4$, the global y now points to 4.
Thus, y is changed.

```python
y = [1,2,3]
def change_by_global():
    global y
    y = 3
print(y)
```

Detour: Understanding Binding

Click for details.

Function Practice

Choose any code we have written (giving grades? printing stars? is string a palindrome?) and turn it to a function.

```
def stars(n):
    for i in range(1, n+1):
        print('*' * i)

stars(int(input()))
```

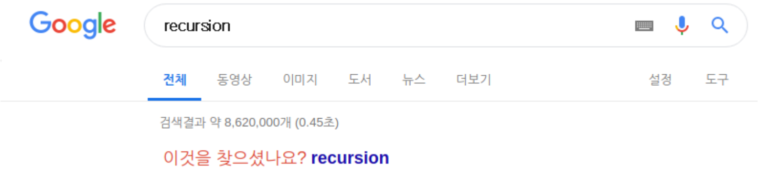
Recursion

What is recursion? It is...

Recursion

What is recursion? It is...

Recursion


Google
recursion
전체
동영상
이미지
도서
뉴스
더보기
설정
도구
검색결과 약 8,620,000개 (0.45초)
이것을 찾으셨나요? recursion

Recursion

A thing is defined in terms of itself or of its type.

Back to highschool

점화식을 일반항으로 변환하기

다음은 등차수열의 점화식입니다.

$$\begin{cases} a(1) = 3 \\ a(n) = a(n-1) + 2 \end{cases}$$

이 점화식에서 다음의 두 가지 정보를 알 수 있습니다:

- 첫째항은 3 입니다
- 이전 항을 구하려면 2를 **더합니다.** 다시 말해서, 공차는 2입니다.

*https://ko.khanacademy.org/

In Python

```python
def a(n):
    # Base Case:
    if n == 1:
        # a(1) = 3
        return 3
    else:
        # a(n) = a(n-1) + 2
        return a(n-1) + 2
```

In English

Assume $a(x)$ will somehow, magically calculate $a(n)$
Then, write out the definition, we are done.

How it works

Calling $a(3)$.

$$
\begin{aligned}
a(3) = {} & a(2) + 2 \\
& \quad a(2) = a(1) + 2 \\
& \qquad\qquad a(1) = 3 \\
& \quad a(2) = 5 \\
a(3) = {} & 7
\end{aligned}
$$

What if $a(1)$ is not defined?

Importance of Base Case

$a(-1) \rightarrow a(-2) \rightarrow a(-3)...$
Forever and ever until we run out of memory.[1]

[1] Or not, look up call stack and tail call optimization

Recursion Practice 0

Write a recursive function that doubles all the element in the list. hint: If a smaller problem(and smaller here means...) is magically solved, what do we do?

```
def doubles(l):
    if len(l) == 0:
        # base case
        return []
    else:
        #FILL ME IN

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

Recursion Practice 1

We used *for* to implement the fibonacci sequence. Rewrite it using recursion.[1]

[1] Church-Turing thesis proves that this is *always* possible, given enough memory. Details here

Solution

```python
def fibonacci(n):
    # fib(0) = 0, fib(1) = 1
    if n == 0 or n == 1:
        return n
    # fib(n) = fib(n-1) + fib(n-2)
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

Food for thought

Does it print *fibonacci*(10) well? What about *fibonacci*(100)?
Compare it to the *for* version. Why is it so slow?

Dynamic Programming: Memoization

It is slow because there is repetitive calculation.
If we can store the calculation results somewhere and check before we start calculation, we can save time. (Tradeoff between memory and time)

DP Solution - Recursive

```python
DEFAULT = -1
tbl = [DEFAULT] * 1000

# tbl[0] = fib(0) = 0, tbl[1] = fib(1) = 1
tbl[0] = 0
tbl[1] = 1

def fibonacci(n):
    # if tbl[n] has been updated
    # use that value instead calculating
    if tbl[n] != DEFAULT:
        return tbl[n]
    # fib(n) = fib(n-1) + fib(n-2)
    else:
        # update the table
        tbl[n] = fibonacci(n-1) + fibonacci(n-2)
        return tbl[n]

print(fibonacci(100))
```

DP Solution - For Ver.

```python
def fibonacci(n):
    fib = [0] * 1000
    fib[0], fib[1] = 0, 1
    for i in range(2, n+1):
        fib[i] = fib[i-1]+fib[i-2]
    return fib[n]

print(fibonacci(int(input())))
```