Python 101

Lec03
Functions

thoum

April 16, 2019

Functions

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

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

Python Functions

```
def f(x):
    r = x+1
    return r

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

#Note that each x is unique to the function

print(f(3)) # f(3) becomes 3+1
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

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;

    if (dummy_args == NULL) {
        if (!(dummy_args = PyTuple_New(0)))
```

Abstraction

- Lets us focus

Understanding Functions

Some functions(*pure* functions) are just values

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

def factorial(n):
    ret = 1
    for i in range(1, n+1):
        ret *= i
    return ret

# please refrain from giving confusing names to
    functions
def triple(x):
    return 4 * x
```

Understanding Functions

Some functions have side effects

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

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

def change_without_global(x):
    x = 2
# no change?!
print(x)
```

Understanding Functions

Some functions have side effects (cont'd)

```
y = [5, 6, 7]
def change_by_global():
    global y
    y = 3
# changed
print(y)
```

Detour: Wait..What just happened?

Function parameters can be thought as names or labels (in python).
Imagine that function is a dark room.
We can only interact with outer objects that enters this room by giving them glow-in-the-dark stickers with their names on it.

Detour: Wait..What just happened?

Here, we gave outer x the sticker that says l.
By l[0], we were able to reach x[0].

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

Detour: Wait..What just happened?

When we $l = 2$, we took the sticker from x and gave it to 2.
Now, we can't see x in the dark.

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

Detour: Wait..What just happened?

Using *global* is like giving a permanent sticker to y, with name y written on it.

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

Detour: Wait..What just happened?

Confusing? Good news: We will get used to it.
Bad news: Side effects are inevitable.
So let's try our best to keep it to a minimum.
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.

Recursion

What is recursion? It is...

Recursion

What is recursion? It is...

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.

Recursion????

What?

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 2**를 더합니다.** 다시 말해서, 공차는 2입니다.

img source

In Python

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

In English

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

In Your Brain

Imagine what happens if we call $a(3)$.

$$
\begin{aligned}
a(3) = & a(2) + 2 \\
& a(2) = a(1) + 2 \\
& \qquad a(1) = a(0) + 2 \\
& \qquad\qquad a(0) = 3 \\
& \qquad a(1) = 5 \\
& a(2) = 7 \\
a(3) = & 9
\end{aligned}
$$

What if $a(0)$ 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

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

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

Solution

```
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

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