

homework questions are not straight forward, project questions are not straight forward, even if you understand what's going on, you could spent a long time..... it's much better to go to office hour to get some help

Iteration Example

- the fibonacci sequence

Discussion Question



Is this alternative definition of `fib` the same or different from the original `fib`?

```
def fib(n):
    """Compute the nth Fibonacci number"""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```



5

Designing Functions

Characteristics of Functions



A function's domain is the set of all inputs it might possibly take as arguments.

A function's range is the set of output values it might possibly return.

A pure function's behavior is the relationship it creates between input and output.

between the input and the output

why do we discuss the domain and the range of the functions here?
i dont see the purpose.

so it is just trytin to explain what functions are for.

A Guide to Designing Function

A Guide to Designing Function



Give each function exactly one job.



not



Don't repeat yourself (DRY). Implement a process just once, but execute it many times.



Define functions generally.



when the world disfined electrical sockets

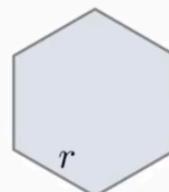
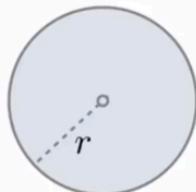
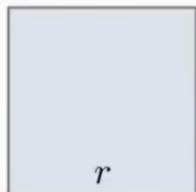
Higher-Order Functions

Generalizing Patterns with Arguments



Regular geometric shapes relate length and area.

Shape:



Area:

$$1 \cdot r^2$$

$$\pi \cdot r^2$$

$$\frac{3\sqrt{3}}{2} \cdot r^2$$

the rest is just how you compute the area of a shape

剩下的就是如何计算形状的面积

11

try to find the similar structure and share the same implementation?

- assert

```
assert (condition), 'error message'
```

```
~/lec$ python3
Python 3.3.1 (v3.3.1:d9893d13c628, Apr  6 2013, 11:07:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> assert 3 > 2, 'Math is broken'
>>> assert 2 > 3, 'That is false'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: That is false
>>> ^D
~/lec$ python3 -i ex.py
>>> area_hexagon(10)
259.8076211353316
>>> area_hexagon(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "ex.py", line 16, in area_hexagon
      return area(r, 3 * sqrt(3) / 2)
  File "ex.py", line 6, in area
    assert r > 0, 'A length must be positive'
AssertionError: A length must be positive
>>>
```

```
1 """Generalization."""
2
3 from math import pi, sqrt
4
5 def area(r, shape_constant):
6     assert r > 0, 'A length must be positive'
7     return r * r * shape_constant
8
9 def area_square(r):
10     return area(r, 1)
11
12 def area_circle(r):
13     return area(r, pi)
14
15 def area_hexagon(r):
16     return area(r, 3 * sqrt(3) / 2)
```



well I'll get an assertion error

我会得到一个断言错误，必须传递一个长度

Generalization of the process



Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

12

```
bash
~/lec$ python3 -m doctest ex.py
~/lec$ python3 -m doctest ex.py
~/lec$ python3 -m doctest ex.py
~/lec$ python3 -m doctest -i ex.py
~/lec$ python3 -m doctest -v ex.py
Trying:
    sum_cubes(5)
Expecting:
    225
ok
Trying:
    sum_naturals(5)
Expecting:
    15
ok
Trying:
    summation(5, cube)
Expecting:
    225
ok
3 items had no tests:
    ex
    ex(cube)
    ex.identity
3 items passed all tests:
    1 tests in ex.sum_cubes
    1 tests in ex.sum_naturals
    1 tests in ex.summation
3 tests in 6 items.
3 passed and 0 failed.
Test passed.
~/lec$
```



```
4     return k
5
6 def cube(k):
7     return pow(k, 3)
8
9 def summation(n, term):
10    """Sum the first N terms of a sequence.
11
12    >>> summation(5, cube)
13    225
14    """
15    total, k = 0, 1
16    while k <= n:
17        total, k = total + term(k), k + 1
18    return total
19
20 def sum_naturals(n):
21    """Sum the first N natural numbers.
22
23    >>> sum_naturals(5)
24    15
25    """
26    return summation(n, identity)
27
28 def sum_cubes(n):
29    """Sum the first N cubes of natural numbers
30
31    >>> sum_cubes(5)
32    225
33    """
34    return summation(n, cube)
```



Summation Example

```
def cube(k):
    return pow(k, 3)
Function of a single argument
(not called term)

def summation(n, term)
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
225
"""
total, k = 0, 1
while k <= n:
    total, k = total + term(k), k + 1
return total
A formal parameter that will
be bound to a function

The cube function is passed
as an argument value

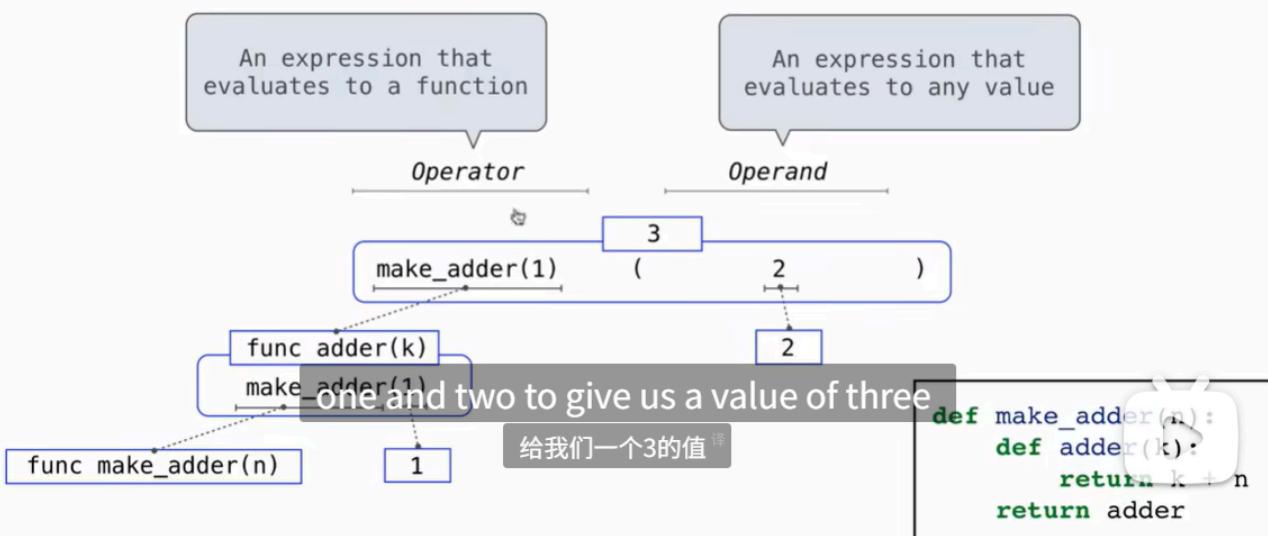
The function bound to term
gets called here
that takes another function as an argument
```

它是一个以另一个函数为参数的函数



| nesting functions???

Call Expressions as Operator Expressions



The Purpose of Higher-Order Functions



Functions are first-class: Functions can be manipulated as values in our programming language.

Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Higher-order functions:

- Express general methods of computation
- Remove repetition from programs
which they take as arguments or return as values
他们作为论据或返回
- Separate concerns among functions



Lambda Expressions

```
>>> x = 10
>>> square = x * x
>>> x
10
>>> square
100
>>> square = lambda x: x * x
>>> square
<function <lambda> at 0x1003c1bf8>
>>> square(4)
16
>>> square(10)
100
>>> (lambda x: x * x)(3)
9
>>>
```



the question is, why would we use the lambda expression if we could just define functions?

Lambda Expressions



```
>>> x = 10
An expression: this one evaluates to a number
>>> square = x * x
Also an expression: evaluates to a function
>>> square = lambda x: x * x
Important: No "return" keyword!
A function
with formal parameter x
that returns the value of "x * x"
in alamda expression
```

需要注意的是，lambda表达式中没有return关键字

lambda functions create simple functions

lambda expressions are not common in Python, but important in general,

especially in some programming languages

| so, in which way?

differences between def and lambda

- only the def statement gives the function an intrinsic name
this affects how we write the frame
 - | however this is only a small difference

Return statement

a return statement completes the evaluation of a call expression and provides its value

return statement within f: switch back to the previous environment

Return Statements



A return statement completes the evaluation of a call expression and provides its value

f(x) for user-defined function f: switch to a new environment; execute f's body

return statement within f: switch back to the previous environment; f(x) now has a value

Only one return statement is ever executed while executing the body of a function

```
def end(n, d):
    """Print the final digits of N in reverse order until D is found.
```

```
>>> end(34567, 5)
7
6
```

let's look at some more examples of one we might return

```
while n > 0:
    last, n = n % 10, // 10
    print(last)
    if d == last:
        return None
```

(Demo)



```
~/lec$ python3 -i ex.py
>>> sqrt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> sqrt = inverse(square)
>>> square(16)
256
>>> sqrt(256)
16
>>> sqrt(16)
4
>>> sqrt(4)
2
>>> sqrt(2)
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ex.py", line 19, in <lambda>
    return lambda y: search(lambda x: f(x) == y)
  File "ex.py", line 6, in search
    x += 1
KeyboardInterrupt
>>>
```

```
search(f):
    x = 0
    while True:
        if f(x):
            return x
        x += 1

def is_three(x):
    return x == 3

def square(x):
    return x * x

def positive(x):
    return max(0, square(x) - 100)

def inverse(f):
    """Return g(y) such that g(f(x)) -> x."""
    return lambda y: search(lambda x: f(x) == y)
```



implementation of square root

所以这不是最理想的平方根实现

19,48

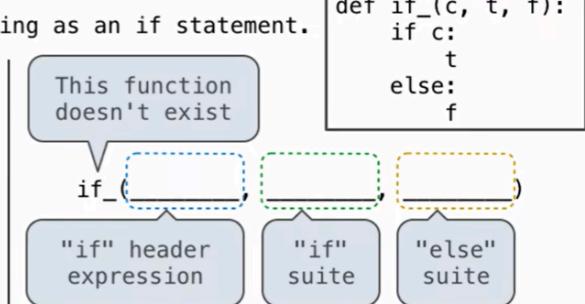
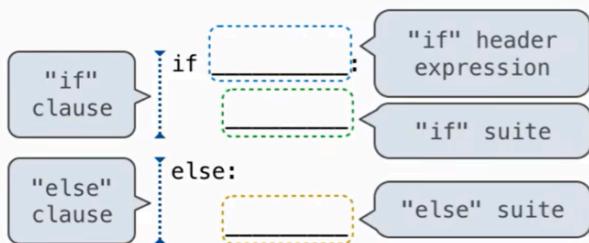
wow i have to say that this is impressive

Control

If Statements and Call Expressions



Let's try to write a function that does the same thing as an if statement.



Execution Rule for Conditional Statements:

Each clause is considered in order.

1. Evaluate the header's expression (if present).

2. If it is a true value (or an else header), execute the suite &

well let's look at a practical example of when it would

execute the suite &

让我们来看看一个实际的例子，说明它何时会在Python中

Evaluation Rule for Call Expressions:

1. Evaluate the operator and then the operand subexpressions

2. Apply the function that is the value of the operator to the arguments that are the values of the operands



(Demo)

8

the same question in Q5 of hw1

```
~/lec$ python3 -i ex.py
>>> real_sqrt(4)
2.0
>>> real_sqrt(-4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ex.py", line 11, in real_sqrt
    return if_(x > 0, sqrt(x), 0.0)
ValueError: math domain error
>>> 
```

```
def if_(c, t, f):
    if c:
        return t
    else:
        return f

from math import sqrt

def real_sqrt(x):
    """The real part of the square root of X."""
    return if_(x > 0, sqrt(x), 0.0)
```



and

这不是调用表达式所能做的



Control Expressions

Logical Operators



To evaluate the expression `<left> and <right>`:

1. Evaluate the subexpression `<left>`.
2. If the result is a false value `v`, then the expression evaluates to `v`.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

To evaluate the expression `<left> or <right>`:

1. Evaluate the subexpression `<left>`.
2. If the result is a true value `v`, then the expression evaluates to `v`.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

let's look at a couple of examples

让我们看几个例子



(Demo)

Conditional Expressions



A conditional expression has the form

`<consequent> if <predicate> else <alternative>`

Evaluation rule:

1. Evaluate the `<predicate>` expression.
2. If it's a true value, the value of the whole expression is the value of the `<consequent>`.
3. Otherwise, the value of the whole expression is the value of the `<alternative>`.

字幕已关闭

```
>>> x = 0
>>> abs(1/x if x != 0 else 0)
0
```