

Curs 6

P00

recapitulare PROGRAMARE PROCEDURALĂ

CHAP.6. FRUITFUL FUNCTIONS

CONTENTS

- ◎ Return values
- ◎ Incremental development
- ◎ Composition
- ◎ Boolean functions
- ◎ More recursion
- ◎ Leap of faith
- ◎ One more example
- ◎ Checking types
- ◎ Debugging, Glossary, Exercises

RETURN VALUES

- ◎ Some of the built-in functions we have used, such as the math functions, have produced **results**.
- ◎ **Calling the function** generates a new value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
```

```
height = radius * math.sin(radians)
```

RETURN VALUES

- ◎ All of the functions we have written so far are **void** – their return value is **None**.
- ◎ Now, we are ready to write **fruitful functions** – functions that return values to the calling function.

e.g.

```
import math
def area (radius):
    temp = math.pi * radius**2
    return temp
```

```
import math
def area (radius):
    return math.pi * radius**2
```



expression

RETURN VALUES

- ⦿ We have seen before the **return statement** without any return value, but **in a fruitful function the return statement includes a return value.**

- ⦿ This statement means:

“Return immediately from this function and use the following expression as a return value.”

- ⦿ As soon as a **return statement** executes, the function terminates **WITHOUT** executing any subsequent statements.

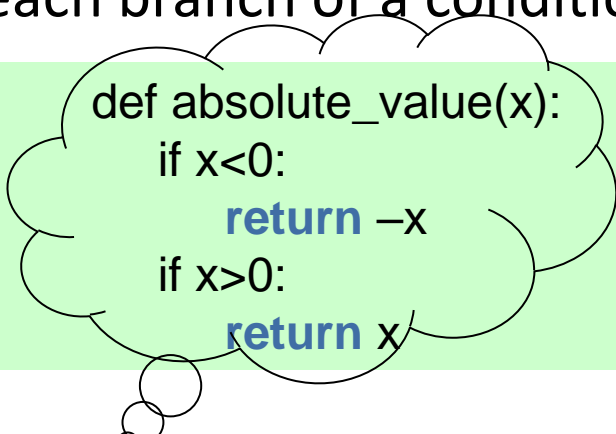
- ⦿ Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

- ⦿ In a fruitful function, it is a good idea to ensure that **every possible path** through the program hits a return statement.

RETURN VALUES

- ◎ Sometimes it is useful to have **multiple return statements**, one in each branch of a conditional.

```
def absolute_value(x):  
    if x<0:  
        return -x  
    else:  
        return x
```



```
def absolute_value(x):  
    if x<0:  
        return -x  
    if x>0:  
        return x
```

- ◎ **The right example is INCORRECT** because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. So, the flow of execution gets to the end of this function and the **return value is None** – which is not the absolute value of 0.

INCREMENTAL DEVELOPMENT

- ◎ The goal of **incremental development** is to avoid long debugging sessions by adding and testing only a small amount of code at a time.
- ◎ The key aspects of the process are:
 1. Start with a working program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
 2. Use temporary variables to hold intermediate values so you can output and check them.

WORK step by step examples from chapter 6.2



INCREMENTAL DEVELOPMENT

- ◎ The key aspects of the process are:
 - 3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but **only if** it does not make the program difficult to read.
 - 4. When you start out, you should add only a line or two of code at a time.
 - 5. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

COMPOSITION

- ◎ One can call one function from within another. This ability is called **composition**.

PROBLEM. Write a function that compute the area of the circle if we have two points: the center and a point of the circle.

```
def circle_area(xc, yc, xp, yp):  
    radius = distance (xc, yc, xp, yp)  
    result = area (radius)  
    return result
```

- ◎ The temporary variables **radius** and **result** are useful for development and debugging, BUT once the program is working, they might be replaced by composing the function calls:

```
def circle_area(xc, yc, xp, yp):  
    return area (distance (xc, yc, xp, yp))
```

BOOLEAN FUNCTIONS

- Functions can return **boolean values**, which is often convenient for hiding complicated tests inside functions.

```
def isDivisible (x, y):  
    if x % y == 0:  
        return True  
    else:  
        return False
```

- We can make the function more concise by taking advantage of the fact that **the condition of the if statement is itself a boolean expression**.

- We can return it directly, avoiding the if statement altogether.

```
def isDivisible (x, y):  
    return x % y == 0
```

MORE RECURSION

frabjuous: An adjective used to describe something that is frabjuous.

- ◎ If you saw that definition in the dictionary, you might be annoyed. This is a **circular definition**.
- ◎ On the other hand, if you looked up the definition of the mathematical function factorial, you might get something like this:

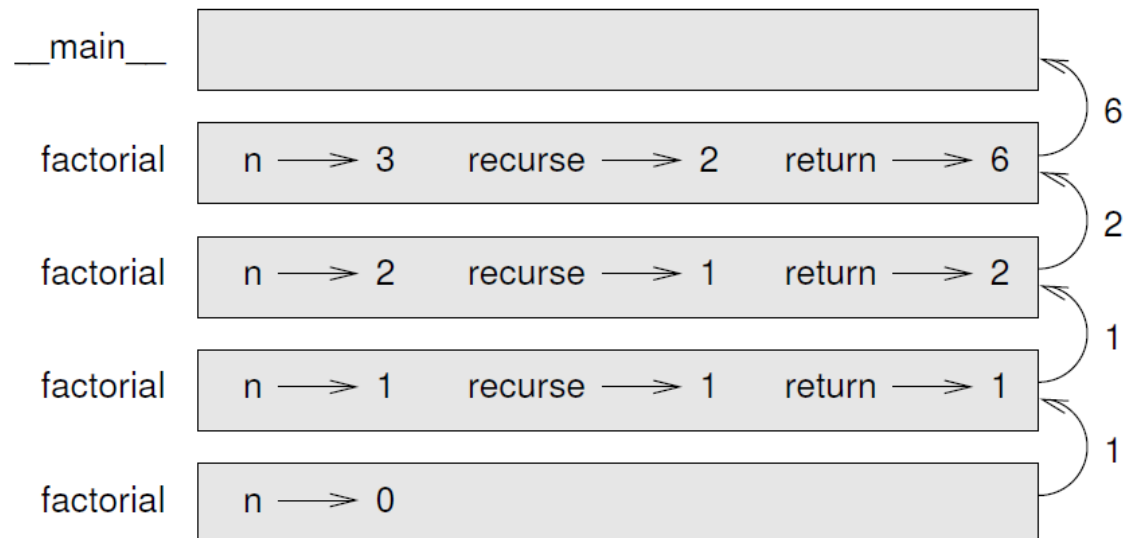
$$0! = 1$$

$$n! = n (n - 1)!$$

- ◎ This definition says that the factorial of 0 is 1, and the factorial of any other value, n , is n multiplied by the factorial of $n - 1$.

MORE RECURSION

```
def factorial (n):  
    if n == 0:  
        return 1  
    else:  
        recurse = factorial (n-1)  
        result = n * recurse  
        return result
```



LEAP OF FAITH

- ◎ Following the flow of execution is one way to read programs, but it can quickly become labyrinthine.

*An alternative is what we call the “**leap of faith**.”*

- ◎ When you come to a function call, instead of following the flow of execution, you **assume** that the function works correctly and returns the appropriate value.

LEAP OF FAITH

- ◎ In fact, you are already practicing this *leap of faith* when you use built-in functions. When you call **math.cos** or **math.exp**, you don't examine the implementations of those functions.
- ◎ You just assume that they work because the people who wrote the built-in functions were good programmers.
- ◎ **The same is true of recursive programs.** When you get to the recursive call, instead of following the flow of execution, *you should assume that the recursive call works* (yields the correct result) and THEN ask yourself, "Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n ?" In this case, it is clear that you can, by multiplying by n .

ONE MORE EXAMPLE

```
def fibonacci (n):  
    if n == 0:  
        return 0  
    elif n==1:  
        return 1  
    else:  
        return fibonacci (n-1) + fibonacci (n-2)
```

CHECKING TYPES

- ◎ What happens if we call `factorial` and give it `1.5` as an argument?

```
>>> factorial (1.5)
```

```
RuntimeError: Maximum recursion depth exceeded
```

- ◎ It looks like an infinite recursion. But how can that be?
- ◎ There is a base case — when `n == 0`. The problem is that *the values of n miss the base case*.
- ◎ In the first recursive call, the value of `n` is `0.5`. In the next, it is `-0.5`. From there, it gets smaller and smaller, but it will never be `0`.

CHECKING TYPES

- ⊙ We can use the built-in function `isinstance(val,type)` to verify the type of the argument.
- ⊙ While we're at it, we also make sure `the argument is positive`.

```
def factorial (n):  
    if not isinstance (n, int):  
        print "Factorial is only defined for integers."  
        return None  
    elif n < 0:  
        print "Factorial is not defined for negative integers."  
        return None  
    elif n == 0:  
        return 1  
    else:  
        return n * factorial (n-1)
```

CHECKING TYPES

- ◎ Test the previous code for

```
>>> factorial("fred")
```

```
>>> factorial(-2)
```

- ◎ This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that *might cause an error*.
- ◎ The **guardians** make it possible to prove the correctness of the code.

The slide features a green background. On the left, there is a vertical bar with a lighter green circle. A horizontal bar with a gradient from light green to white spans the width of the slide, positioned below the title.

Debugging, Glossary, Exercises