Chapter 6
Return Values

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

- 1. Some Functions Have Return Values
- 2. And Some Have None
- 3. Return Values and Conditionals
- 4. Incremental Development
- 5. Boolean Functions
- 6. Recursion with Return Values
- 7. Fibonacci
- 8. Checking Types

1. Some Functions Have Return Values

- When you call a function like **math.sqrt**, the result is called a **return value**. If the function call appears at the end of a cell, Jupyter displays the return value immediately:

```
1 import math
2
3 math.sqrt(42 / math.pi)
4
```

```
3.656366395715726
```

- If you assign the return value to a variable, it doesn't get displayed:

```
1 import math
2
3 radius = math.sqrt(42 / math.pi)
4
```

But you can display it later:

```
1 radius
```

```
3.656366395715726
```

1. Some Functions Have Return Values

- Or you can use the return value as part of an expression:

```
1 import math
2
3 radius + math.sqrt(42 / math.pi)
4
```

```
7.312732791431452
```

1. Some Functions Have Return Values

- An example of a function that returns a value:

```
1 import math
2
3 def circle_area(radius):
4     area = math.pi * radius**2
5     return area
```

  - The function **circle_area** takes **radius** as a parameter and computes the area of a circle with that radius.
  - The last line is a **return** statement that returns the value of area.

- If we call the function like this, Jupyter displays the return value:

```
1 circle_area(5)
```

```
78.53981633974483
```

1. Some Functions Have Return Values

- We can assign the return value to a variable or use it as part of an expression:

```
1 a = circle_area(radius)
```

```
1 circle_area(radius) + 2 * circle_area(radius / 2)
```

```
63.000000000000014
```

2. And Some Have None

- If a function doesn't have a **return** statement, it returns **None**, which is a special value.

```
1 def repeat(word, n):
2     print(word * n)
```

- This function uses the **print** function to display a string, but it does not use a **return** statement to return a value. If we assign the result to a variable, and if we display the value of the variable, we get nothing:

```
1 result = repeat('Finland, ', 3)
```

```
Finland, Finland, Finland,
```

```
1 result
```

2. And Some Have None

- **result** actually has a value, but Jupyter doesn't show it. However, we can display it like this:

```
1 def repeat(word, n):
2     print(word * n)
```

```
1 result = repeat('Finland, ', 3)
```

```
Finland, Finland, Finland,
```

```
1 print(result)
```

```
None
```

- The return value from **repeat** is None.

2. And Some Have None

- A function similar to **repeat** except that it has a return value:

```
1 def repeat_string(word, n):
2     return word * n
```

```
1 line = repeat_string('Spam, ', 4)
2 print(line)
```

```
Spam, Spam, Spam, Spam,
```

  - Notice that we can use an expression in a **return** statement, not just a variable.

2. And Some Have None

```
1 def repeat(word, n):
2     print(word * n)
```

```
1 def repeat_string(word, n):
2     return word * n
```

- A function like this **repeat_string** is called a **pure function** because it doesn't display anything or have any other effect—other than returning a value.

3. Return Values and Conditionals

- We can write the **abs** function like this:

```
1 def absolute_value(x):
2     if x < 0:
3         return -x
4     else:
5         return x
```

  - If **x** is negative, the first return statement returns **-x** and the function ends immediately. Otherwise, the second **return** statement returns **x** and the function ends. So this function is correct.

3. Return Values and Conditionals

- However, if you put **return** statements in a conditional, you have to make sure that every possible path through the program hits a **return** statement. For example, here's an **incorrect version** of **absolute_value**:

```
1 def absolute_value_wrong(x):
2     if x < 0:
3         return -x
4     if x > 0:
5         return x
```

  - What happens if we call this function with 0 as an argument?

```
1 absolute_value_wrong(0)
```

    - We get nothing! Here's the problem: when **x** is 0, neither condition is true, and the function ends without hitting a return statement, which means that the return value is **None**.

3. Return Values and Conditionals

- Here's a version of **absolute_value** with an extra **return** statement at the end:

```
1 def absolute_value_extra_return(x):
2     if x < 0:
3         return -x
4     else:
5         return x
6     return 'This is dead code'      # This code cannot run -> dead code
```

  - If **x** is negative, the first **return** statement runs and the function ends. Otherwise the second **return** statement runs and the function ends. Either way, we never get to the third **return** statement—so it can never run.

- Code that can never run is called **dead code**.
  - In general, dead code doesn't do any harm, but it often indicates a misunderstanding, and it might be confusing to someone trying to understand the program.

4. Incremental Development

- As you write larger functions, you might find yourself spending more time debugging. To deal with increasingly complex programs, you might want to try **incremental development**, which is a way of adding and testing only a small amount of code at a time.
- As an example, suppose you want to find the distance between two points represented by the coordinates (x_1, y_1) and (x_2, y_2) . By the Pythagorean theorem, the distance is:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

4. Incremental Development

- The first step is to consider what a **distance** function should look like in Python—that is, what are the **inputs** (parameters) and what is the **output** (return value)?
  - For this function, the inputs are the coordinates of the points. The return value is the distance. Immediately you can write an outline of the function:

```
1 def distance(x1, y1, x2, y2):
2     return 0.0
```

- This version doesn't compute distances yet—it always returns zero. But it is a complete function with a return value, which means that you can test it before you make it more complicated.
  - To test the new function, we'll call it with sample arguments:

```
1 distance(1, 2, 4, 6)
```

```
0.0
```

4. Incremental Development

- At this point we have confirmed that the function runs and returns a value, and we can start adding code to the body.
  - A good next step is to find the differences x_2 - x_1 and y_2 - y_1. Here's a version that stores those values in temporary variables and displays them:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

  - Test the function with the sample arguments.
    - If the function is working, it should display **dx** is 3 and **dy** is 4 . If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

```
distance(1, 2, 4, 6)
```

```
dx is 3
dy is 4
0.0
```

4. Incremental Development

- Good so far. Next we compute the sum of squares of **dx** and **dy**:

```
1 def distance(x1, y1, x2, y2):
2     dx = x2 - x1
3     dy = y2 - y1
4     dsquared = dx**2 + dy**2
5     print('dsquared is: ', dsquared)
6     return 0.0
```

  - Again, we can run the function and check the output, which should be 25:

```
1 distance(1, 2, 4, 6)
```

```
dsquared is:  25
0.0
```

4. Incremental Development

- Finally, we can use **math.sqrt** to compute the distance:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    print("result is", result)
    return 0.0
```

  - And test it:

```
distance(1, 2, 4, 6)
```

```
result is 5.0
0.0
```

4. Incremental Development

- The result is correct. We remove the **print** statement and return the correct value:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

  - This version of **distance** is a pure function. If we call it like this, only the result is displayed:

```
distance(1, 2, 4, 6)
```

```
5.0
```

4. Incremental Development

- The **print** statements we wrote are useful for debugging, but once the function is working, we can remove them. Code like that is called **scaffolding** because it is help ful for building the program but is not part of the final product.
- This example demonstrates incremental development. The key aspects of this process are:
  - 1. Start with a working program, make small changes, and test after every change.
  - 2. Use variables to hold intermediate values so you can display and check them.
  - 3. Once the program is working, remove the scaffolding.
- At any point, if there is an error, you should have a good idea where it is. Incremental development can save you a lot of debugging time.

5. Boolean Functions

- Functions can return the boolean values **True** and **False**, which is often convenient for encapsulating a complex test in a function. For example, **is_divisible** checks whether **x** is divisible by **y** with no remainder:

```
1 def is_divisible(x, y):
2     if x % y == 0:
3         return True
4     else:
5         return False
```

```
1 def is_divisible(x, y):
2     return x % y == 0
```

  - Here's how we use it:

```
1 is_divisible(6, 4)
```

```
False
```

5. Boolean Functions

- Boolean functions are often used in conditional statements:

```
1 if is_divisible(6, 2):
2     print('divisible')
```

```
divisible
```

or

```
1 if is_divisible(6, 2) == True:
2     print('divisible')
```

```
divisible
```

  - But the comparison is unnecessary.

6. Recursion with Return Values

- Now that we can write functions with return values, we can write recursive functions with return values, and with that capability, we have passed an important threshold—the subset of Python we have is now **Turing complete**, which means that we can perform any computation that can be described by an algorithm.
- To demonstrate recursion with return values, we'll evaluate a few recursively defined mathematical functions.
- A recursive definition is similar to a circular definition, in the sense that the definition refers to the thing being defined. A truly circular definition is not very useful:
  - vorpal: An adjective used to describe something that is vorpal.

  If you saw that definition in the dictionary, you might be annoyed.

6. Recursion with Return Values

- Definition of the factorial function:

  $0! = 1$

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

  – 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 $n - 1$.

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         recurse = factorial(n-1)
6         return n * recurse
```

```
1 factorial(3)
```

```
6
```

6. Recursion with Return Values

- The flow of execution for **factorial(3)**:

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         recurse = factorial(n-1)
6         return n * recurse
```

```
1 factorial(3)
```

```
6
```

- Since 3 is not 0, we take the second branch and calculate the factorial of *n*-1...
  - Since 2 is not 0, we take the second branch and calculate the factorial of *n*-1...
    - Since 1 is not 0, we take the second branch and calculate the factorial of *n*-1...
      - Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.
    - The return value, 1, is multiplied by *n*, which is 1, and the result is returned.
  - The return value, 1, is multiplied by *n*, which is 2, and the result is returned.
- The return value 2 is multiplied by *n*, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

6. Recursion with Return Values

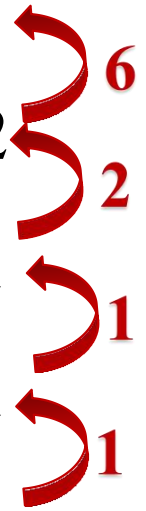
- The stack diagram for **factorial(3)**:

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         recurse = factorial(n-1)
6         return n * recurse
```

```
1 factorial(3)
```

```
6
```

- __main__
- factorial $n \rightarrow 3$ recurse $\rightarrow 2$
- factorial $n \rightarrow 2$ recurse $\rightarrow 1$
- factorial $n \rightarrow 1$ recurse $\rightarrow 1$
- factorial $n \rightarrow 0$



7. Fibonacci

- https://tuoitre.vn/day-so-fibonacci-va-nhung-bi-an-trong-tu-nhien-20180313151043875.htm
- In mathematics, the Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones.
- Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers, commonly denoted F_n.
- Many writers begin the sequence with 0 and 1, although some authors start it from 1 and 1 and some (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the sequence begins:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

7. Fibonacci

- fibonacci(0) = 0
- fibonacci(1) = 1
- fibonacci(n) = fibonacci($n - 1$) + fibonacci($n - 2$)

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

```python
fibonacci(5)
```

8. Checking Types

- What happens if we call **factorial(1.5)**?
  - In this example, the initial value of **n** is 1.5. In the first recursive call, the value of **n** is 0.5.
  - In the next, it is -0.5. From there, it gets smaller (more negative), **but it will never be 0**.
  - To avoid infinite recursion we can use the built-in function **isinstance** to check the type of the argument.

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         recurse = factorial(n-1)
6         return n * recurse
```

```
1 factorial(1.5)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[6], line 1
----> 1 factorial(1.5)

Cell In[5], line 5, in factorial(n)
      3     return 1
      4 else:
----> 5     recurse = factorial(n-1)
      6     return n * recurse

Cell In[5], line 5, in factorial(n)
      3     return 1
      4 else:
----> 5     recurse = factorial(n-1)
      6     return n * recurse

    [... skipping similar frames: factorial at line 5 (2975 times)]

Cell In[5], line 5, in factorial(n)
      3     return 1
      4 else:
----> 5     recurse = factorial(n-1)
      6     return n * recurse

RecursionError: maximum recursion depth exceeded
```

8. Checking Types

– Here's how we check whether a value is an integer:

```
1 isinstance(3, int)
```

```
True
```

```
1 isinstance(3.5, int)
```

```
False
```

8. Checking Types

- Now here's a version of **factorial** with error checking:

```
1  def factorial(n):
2      if not isinstance(n, int):
3          print('factorial is only defined for integers.')
4          return None
5      elif n < 0:
6          print('factorial is not defined for negative numbers.')
7          return None
8      elif n == 0:
9          return 1
10     else:
11         return n * factorial(n-1)
```

```
1  factorial(1.5)
```

```
factorial is only defined for integers.
```

- Checking the parameters of a function to make sure they have the correct types and values is called **input validation**.