Think Python 2e, Chapter 6 Notes

Geoffrey Matthews

September 14, 2022

Fruitful Functions

```python
def area(radius):
    a = math.pi * radius**2
    return a
```

Fruitful Functions

```python
def area(radius):
    a = math.pi * radius**2
    return a
```

Return value can be complex:

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

But easier to debug with **temporary variables**.

More than one return

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

Only one return executes.

More than one return

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

Only one return executes.
Make sure every possible path through the function hits a return!

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

What does it return if `x == 0`?

Incremental development

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

Can immediately write:

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

Incremental development

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

Can immediately write:

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

- Not much, but at least it runs without error and returns a value of the correct type.
- If this is part of a much larger program that computes distances, does arithmetic with them, prints them out, *etc.* you can check the rest of the program without finishing this part.

Incremental development

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

Start adding some intermediate values:

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

- Develop a test case, where you know the answer.
- For example: `distance(0, 0, 3, 4)`
- I know this is 5 because a 3,4,5 triangle is a right triangle.
- Testing the above we can make sure that `dx` is 3 and `dy` is 4

Incremental development

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

Add some more code:

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

Test it.

Incremental development

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

Finally, add the last bit of code:

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

Test this. If all is well, we're done.

Using intermediate variables allows us test each step.

such print statements are called **scaffolding**.

Alternative strategy: dead code

```python
def distance(x1,y1, x2,y2):
    return 5;
    dx = x2 - x1
    dy = y2 - y1
    dsq = dx**2+dy**2
    result = math.sqrt(dsq)
    return result
```

Alternative strategy: dead code

```python
def distance(x1,y1, x2,y2):
    return 5;
    dx = x2 - x1
    dy = y2 - y1
    dsq = dx**2+dy**2
    result = math.sqrt(dsq)
    return result
```

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

Alternative strategy: dead code

```python
def distance(x1,y1, x2,y2):
    return 5;
    dx = x2 - x1
    dy = y2 - y1
    dsq = dx**2+dy**2
    result = math.sqrt(dsq)
    return result
```

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

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

Alternative strategy: dead code

```python
def distance(x1,y1, x2,y2):
    return 5;
    dx = x2 - x1
    dy = y2 - y1
    dsq = dx**2+dy**2
    result = math.sqrt(dsq)
    return result
```

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

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

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

Debugging modules

```python
import math

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

# The following code will be run if this module is run
# It will NOT be run if the module is imported
if __name__ == ’__main__’:
    print(distance(0,0,3,4), ’should be’, 5)
    print(distance(3,4,0,0), ’should be’, 5)
    print(distance(1,1,4,5), ’should be’, 5)
    print(distance(0,0,6,8), ’should be’, 10)
```

Boolean functions

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

Frequently given names that sound like yes/no questions and their answers:

- Is x divisible by y?
- Yes, x is divisible by y.

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

Boolean functions

Frequently boolean functions can be shortened:

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

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

Circular definitions

vorpal: an adjective used to define something that is vorpal.

Recursive definitions

vorpal: an adjective applying to either:

- a cat
- a dog
- a box containing something vorpal

Recursive functions

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$$

Recursive functions

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$$

```python
def factorial(n):
    if n < 0:
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Recursive functions

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

```python
factorial(3)
```

__main__				
factorial	n→3	recurse→2	result→6	↑6
factorial	n→2	recurse→1	result→2	↑2
factorial	n→1	recurse→1	result→1	↑1
factorial	n→0			↑1

Fibonacci numbers

$$fibonacci(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fibonacci(n-1) + fibonacci(n-1) & \text{otherwise} \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$$

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

Checking arguments

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

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
>>> factorial(-3)
RuntimeError: Maximum recursion depth exceeded
```

Checking arguments

```python
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)
```

The two conditionals that return `None` are called **guardians**.

Vocabulary

temporary variable: A variable used to store an intermediate value in a complex calculation.

dead code: Part of a program that can never run, often because it appears after a return statement.

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

scaffolding: Code that is used during program development but is not part of the final version.

guardian: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

Ackermann function

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1, A(x,y-1)) & \text{otherwise} \end{cases}$$

Ackermann function

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1,A(x,y-1)) & \text{otherwise} \end{cases}$$

A	y														
x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
2	3	5	7	9	11	13	15	17	19	21	23	25	27	29	...
3	5	13	29	61	125	253	509	1021	2045	4093	8189	16381	32765	65533	...
4	13	65533	?	?	?	?	?	?	?	?	?	?	?	?	...
5	65533	?	?	?	?	?	?	?	?	?	?	?	?	?	...
6	?	?	?	?	?	?	?	?	?	?	?	?	?	?	...
7	?	?	?	?	?	?	?	?	?	?	?	?	?	?	...
8	?	?	?	?	?	?	?	?	?	?	?	?	?	?	...
9	?	?	?	?	?	?	?	?	?	?	?	?	?	?	...