09-trace-debug

February 13, 2022

0.1 Code tracing and debugging

Once you come up with an idea for a code, it is almost never the case that your first implementation will work. When this happens, you need to take a breath and start debugging.

Debugging is basically tracing your program in your head (often with the help of some pen and paper) for some particular input, and finding out where and why it fails. Obviously you should choose an input that makes your program fail, but something small enough so that you don't need to run too many steps until failing. Your input should be **as simple as possible and as complicated as necessary**.

If you happen to have a computer around, you can use it to help you trace the code by adding `print` statements to your code.

[1]:
```
print("Hello world!")
```

```
Hello world!
```

`print` is a python command used to print things to the console.

Whatever is in the parentheses is going to be printed. If the thing in parentheses is in quotes, it is going to be printed *ipsis litteris* (word per word). If the thing in parentheses is a variable, python prints the value that is inside that variable.

[2]:
```
x = 10
print(x)
```

```
10
```

Quotes and variables can be combined inside prints to make the messages more understandable. A comma indicates there is a space between the parts.

[3]:
```
x = 42
print("The answer to the life, universe, and everything is", x, ".")
```

```
The answer to the life, universe, and everything is 42 .
```

Note that this does not alter the flow of your program. Python does nothing to whatever is printed, and this is only for your scrutinity.

You can use `print` to find out how many times a loop runs, and what is the value of the loop variable.

[4]:
```
def f(n):
    for i in range(n):
        print("i is", i)

f(3)
```

```
i is 0
i is 1
i is 2
```

[5]:
```
def f(n):
    for i in range(1,n):
        print("i is", i)

f(3)
```

```
i is 1
i is 2
```

[6]:
```
def f(n):
    for i in range(1, n + 1):
        print("i is", i)

f(3)
```

```
i is 1
i is 2
i is 3
```

`print`s are particularly useful if there are loops inside loops.

[7]:
```
def f(n, m):
    for i in range(n):
        print(i, ": ", end="") # end="" avoids a line break after print

        for j in range(m):
            print(j, end=", ") # end=", " places a comma after every print
        print()

f(4, 3)
```

```
0 : 0, 1, 2,
1 : 0, 1, 2,
2 : 0, 1, 2,
3 : 0, 1, 2,
```

[8]:
```
def f(n):
    for i in range(n):
```

```
        print(i, ": ", end="")

        # Range for the second loop depends on first loop variable
        for j in range(i):
            print(j, end=", ")
        print()

f(10)
```

```
0 :
1 : 0,
2 : 0, 1,
3 : 0, 1, 2,
4 : 0, 1, 2, 3,
5 : 0, 1, 2, 3, 4,
6 : 0, 1, 2, 3, 4, 5,
7 : 0, 1, 2, 3, 4, 5, 6,
8 : 0, 1, 2, 3, 4, 5, 6, 7,
9 : 0, 1, 2, 3, 4, 5, 6, 7, 8,
```

0.1.1 Fibonacci

The Fibonacci series is a famous series of numbers defined as:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

So, the zero-th Fibonacci number is 1, the first is 1, the second is 2 (1+1), the third is 3 (1+2), and so on and so forth.

Fibonacci is related to the *golden ratio*, and it also appears in several patterns in nature, such as in flowers and pinecones. If you would like to know more about it, check out its Wikipedia page.

Implement the function `fibonacci(n)` that returns the n-th Fibonacci number. For example, `fibonacci(4)` should return 5.

[9]:
```
# Debug and fix this code
def fibonacci(n):
    f0 = 1
    f1 = 1

    for i in range(n):
        next = f0 + f1
        f0 = f1
        f1 = next

    return next
```

```
# assert(fibonacci(0) == 1)
# assert(fibonacci(1) == 1)
# assert(fibonacci(2) == 2)
# assert(fibonacci(3) == 3)
# assert(fibonacci(4) == 5)
```

0.1.2 Exercise 1

Write the function `printNumberTriangle` that takes one integer `n` and prints out a number triangle based on `n`. For example, given the number 4, this function would print out:

```
1
21
321
4321
```

[10]:
```
def printNumberTriangle(n):
    return None # For functions whose return value we do not care about
```

0.1.3 Exercise 2

Print all numbers between `x` and `y`, not inclusive, whose remainder when dividing it by 5 is equal to 2 or 3. Return how many numbers you have printed.

[11]:
```
def rest23(x, y):
    return 42
```

0.1.4 Exercise 3

You are writing the positive integers in increasing order starting from one. But you have never learned the digit zero, and thus omit any number that contains a zero in any position. The first ten integers you write are: 1, 2, 3, 4, 5, 6, 7, 8, 9, and 11. You have just written down the integer `n` (which is guaranteed to not contain the digit zero). What will be the next integer that you write down?

Implement the function `forbiddenZero(n)` that returns the next integer you are going to write after `n`.

[12]:
```
def forbiddenZero(n):
    return 42
```

0.1.5 Exercise 4

Implement the function `triangle(n)` that returns nothing, but prints on the screen a triangle of stars with `n` rows. For example, `triangle(5)` should print:

```
    *
   * *
  * * *
 * * * *
```

```
* * * * *
```

[13]:
```
def triangle(n):
    return None
```