Lecture 4: Intended learning outcomes

By the end of this lecture, you will be able to:

- identify different types of bugs in programs;
- minimize the chance of bugs appearing;
- debug programs.

4 Debugging

4.1 Bugs and debugging

A *bug* is something which stops your program from working as it should. Sometimes such a thing is just called an *error*. Bugs seem to be an inevitable aspect of programming!

Debugging is finding and fixing bugs. This is a fundamental part of programming. Debugging is like detective work; it can be more like an art than a science. The more you do it, the better you will get.

4.2 Types of bugs

Bugs can roughly be split into three types. These are, in order of difficulty to fix, the following:

- Syntax errors;
- Runtime errors (also called exceptions);
- Semantic errors (also called logic errors).

4.3 Syntax errors

These are the easiest errors to find and fix. Python will refuse to run. Spyder will identify syntax errors. They are usually caused by a typographical mistakes, such as:

- a missing bracket or quotation mark;
- a spelling mistake.

For example:

```
pront("Hello there!)
```

4.4 Runtime errors (exceptions)

These are the errors that cause Python to stop in the middle of a program. You are usually asking Python to do something it cannot do. For example:

- divide by zero;
- operate on an undefined variable.

Note that the problem is not always where Python stops. The source of the problem could be elsewhere.

```
a = int(input("Enter a: "))
b = a//3
print("3/a =", 1/b)
```

How do we find the error here?

4.5 Semantic errors (logic errors)

These are the hardest bugs to find. The program runs without complaining about errors. However, the output is not what you had intended. The computer is doing what it is told to do. You just told it to do the wrong thing!

```python
def square(a):
    return a^2

for i in range(7):
    print(i, "^2 + 3 = ", square(i) + 3, sep="")
```

What happens when we run this? How do we find the problem?

4.6 Minimizing bugs

Whilst you will inevitably find bugs, there are ways to reduce the chance of them appearing, or making them easier to find if they do appear.

1 Understand the task before starting to code.
2 Write your code incrementally.
3 Save different versions of your program.
4 Use meaningful variable names.
5 Try to simplify your code where possible.
6 Use functions to simplify your code.
7 Make the flow of your code obvious (e.g. don't use **`break`**).
8 Create a selection of test cases to check your program with.

4.7 Debugging techniques

There are some standard ways of trying to hunt down bugs.

1. Add print statements to reveal values of variables.
2. Isolate where you think the problem lies.
3. Comment out statements which might be causing the problem.
4. Revert to an earlier working version add the alterations slowly.
5. Talk through the code line-by-line to an inanimate object.
   "Rubber duck debugging."

4.8 Rubber duck debugging

This is a very important and useful technique, which might seem daft! The idea is that by explaining your problem *very carefully* to someone else you can often understand where the problem lies yourself. Importantly, the someone else can be some*thing* else. Something else like a rubber duck, or a teddy bear, or a glove puppet!

The key is to go through your program explaining how it works *in detail*. This is why we ask on Blackboard for you to describe your problem in detail.

4.9 Debug this

```python
# Sieve of Eratosthenes: takes the list of numbers
# from 2 to n, then for y from 2 to sqrt(n), removes
# multiples of y.
n = 10000
numbers = list(range(2, n+1))
a = 2
while a <= n**0.5:
    for b in numbers:
        while numbers.count(b) == 1:
            if b % a == 0 and b / a != 1:
                numbers.remove(b)
        a = a + 1
print(numbers)
print("\nThere were", len(numbers),
      "prime numbers up to", n)
```

Find the errors in this program and identify the kind of error.

```
my_string = input("Enter a string. ")

length = len(my_string)

if len < 10:
     print("Your string has more than 10 characters.)
else:
     print("Your string has at most 10 characters.")
```

Do the same for this program.

```
# The user is asked to enter a number (N) less than 20.
# If the number is greater than or equal to 20 or less
# than zero, the program asks again for the number. If
# the number is smaller than 20 and bigger than -1, the
# computer prints out the numbers between N and 0, in
# steps of -1.

N = -1

while (N >= 20 and N < 0):
     N = input("Please enter a number: ")

while N >= 0:
     print(N)
N = N + 1
```