1. Basic Syntax

Based on Practical Python Programming

Example problem to get us acquainted with the syntax:

> One morning, you go out and place a dollar bill on the sidewalk by the Sears tower in Chicago. Each day thereafter, you go out double the number of bills. How long does it take for the stack of bills to exceed the height of the tower?"

Given as known bill thickness and the tower height, a program that solves that problem is as follows:

```python
# sears.py
bill_thickness = 0.11 * 0.001 # Meters (0.11 mm)
sears_height = 442 # Height (meters)
num_bills = 1
day = 1

while num_bills * bill_thickness < sears_height:
        print(day, num_bills, num_bills * bill_thickness)
        day = day + 1
        num_bills = num_bills * 2

print('Number of days', day)
print('Number of bills', num_bills)
print('Final height', num_bills * bill_thickness)
```

```
PS C:\dev\GameAI>python tower.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
...
19 262144 28.83584
20 524288 57.67168
21 1048576 115.34336
22 2097152 230.68672
Number of days 23
Number of bills 4194304
Final height 461.37344
```

Statements

A python program is a sequence of statements separated by a newline. No need for semicolons!

```
num_bills = 1
day = 1
...
        num_bills = num_bills * 2
...
print('Number of days', day)
```

Comments

Comments are denoted by # and are text that will not be executed. They extend to the end of the line.

```
# Meters (0.11 mm)
bill_thickness = 0.11 * 0.001
sears_height = 442 # Height (meters)
```

Variables

Variables store data in memory and can be named using every letter (a-z,A-Z) as well as underscore `_`. They can also contain numerical digits, as long as they don't start with one.

```
height = 442 # valid
_height = 442 # valid
height2 = 442 # valid
2height = 442 # invalid
```

Python is case sensitive. Upper and lower-case letters are considered different letters. These are all different variables:

```
name = 'Jake'
Name = 'Elwood'
NAME = 'Guido'
```

Language statements are always lower-case.

```
while x < 0:   # OK
WHILE x < 0:   # ERROR
```

Looping

The `while` statement executes a loop.

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2
```

The statements indented below the `while` will execute as long as the expression after the `while` is `true`. Remember to put a colon `:` after the while condition!

If we wanted to repeat the loop a fixed amount of times, we could have used a `for` loop.
We use for loops in combination with iterators, a more advanced concept. Just know for now that using `for i in range(n)` will repeat a loop n times.

```
n_iterations = 10
for i in range(n_iterations):
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2
```

The statements indented below the `for` will execute the loop `n_iterations` times.

Indentation

Indentation is used to denote groups of statements that go together. This is similar to using curly brackets `{}` for code blocks in other languages. Consider the previous example:

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Number of days', day)
```

Indentation groups the following statements together as the operations that repeat:

```
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2
```

Because the `print()` statement at the end is not indented, it does not belong to the loop. The empty line is just for readability. It does not affect the execution.

Conditionals

The `if` statement is used to execute a conditional:

```
if a > b:
    print('Computer says no')
else:
    print('Computer says yes')
```

You can check for multiple conditions by adding extra checks using `elif`.

```
if a > b:
    print('Computer says no')
elif a == b:
    print('Computer says yes')
else:
    print('Computer says maybe')
```

Printing

The `print` function produces a single line of text with the values passed.

```
print('Hello world!') # Prints the text 'Hello world!'
```

You can use variables. The text printed will be the value of the variable, not the name.

```
x = 100
print(x) # Prints the text '100'
```

If you pass more than one value to `print` they are separated by spaces.

```
name = 'Jake'
print('My name is', name) # Print the text 'My name is Jake'
```

`print()` always puts a newline at the end.

```
print('Hello')
print('My name is', 'Jake')
```

This prints:

```
Hello
My name is Jake
```

The extra newline can be suppressed:

```
print('Hello', end=' ')
print('My name is', 'Jake')
```

This code will now print:

```
Hello My name is Jake
```

User input

To read a line of typed user input, use the `input()` function:

```
name = input('Enter your name:')
print('Your name is', name)
```

`input` prints a prompt to the user and returns their response. This is useful for small programs, learning exercises or simple debugging. It is not widely used for real programs.

Functions

Use functions for code you want to reuse. Here is a function definition:

```
def sumcount(n):
    '''
    Returns the sum of the first n integers
    '''
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

To call a function.

```
a = sumcount(100)
```

A function is a series of statements that perform some task and return a result. The `return` keyword is needed to explicitly specify the return value of the function.

Exercise 1.5: The Bouncing Ball

> A rubber ball is dropped from a height of 100 meters and each time it hits the ground, it bounces back up to 3/5 the height it fell. Write a program `bounce.py` that prints a table showing the height of the first 10 bounces.

Your program should make a table that looks something like this:

```
1 60.0
2 36.0
3 21.599999999999998
4 12.959999999999999
5 7.775999999999999
6 4.6655999999999995
7 2.7993599999999996
8 1.6796159999999998
9 1.0077695999999998
10 0.6046617599999998
```

Note: You can clean up the output a bit if you use the round() function. Try using it to round the output to 4 digits.

```
1 60.0
2 36.0
3 21.6
4 12.96
5 7.776
6 4.6656
7 2.7994
8 1.6796
9 1.0078
10 0.6047
```

Exercise 1.6: Debugging

The following code fragment contains code from the Sears tower problem. It also has a bug in it.

```python
# sears.py

bill_thickness = 0.11 * 0.001    # Meters (0.11 mm)
sears_height   = 442             # Height (meters)
num_bills      = 1
day            = 1

while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = days + 1
    num_bills = num_bills * 2

print('Number of days', day)
print('Number of bills', num_bills)
print('Final height', num_bills * bill_thickness)
```

Copy and paste the code that appears above in a new program called `sears.py`. When you run the code you will get an error message that causes the program to crash like this:

```
Traceback (most recent call last):
  File "sears.py", line 10, in <module>
    day = days + 1
NameError: name 'days' is not defined
```

Reading error messages is an important part of Python code. If your program crashes, the very last line of the traceback message is the actual reason why the the program crashed. Above that, you should see a fragment of source code and then an identifying filename and line number.

- Which line is the error?
- What is the error?
- Fix the error
- Run the program successfully