Introduction to Programming with Python Documentation

Release 2016.04.26

OpenTechSchool

November 04, 2016

Contents

CHAPTER 1

Getting started

1.1 What you'll need

1.1.1 A Python!

If you haven't yet got python, the latest official installation packages can be found here:

http://python.org/download/

Python 3 is preferable, being the newest version out!

Note: On Windows, you'll need to add **`Python`** to your `%PATH%`, so it can be found by other programs.

When installing Python 3.5 or later, there should be tick box option to do this on the first page of the installer. Make sure you tick this on.

Otherwise, you can run the script under `ToolsScriptswin_add2path.py` where you installed Python.

1.1.2 And a Code Editor

A code editor helps with reading and writing programming code. There are many around, and it is one of the most personal choices a programmer can make - Like a tennis-player choosing their racket, or a chef choosing their favourite knife. To start off with, you'll just want a basic, easy-to-use one that doesn't get in your way, but is still effective at writing python code. Here are some suggestions for those:

- Atom: Windows, Mac & Linux. A new code editor made by Github. It's an open-source project and is very easy to add functionality for, with its packages system.
- Sublime Text: Windows, Mac & Linux. A great all around editor that's easy to use. It's Ctl+B shortcut lets you run the python file you're working on straight away.
- Geany: Windows, Mac & Linux. A simple editor that doesn't aim to be too complicated.
- TextMate: Mac. One of the most famous code editors for Mac, it used to be a paid product but has since been open-sourced.
- Gedit and Kate: Linux. If you run Linux using Gnome or KDE respectively, you might already have one of these two installed!
- Komodo Edit: Windows, Mac & Linux. a sleak, free editor based on the more powerful Komodo IDE.

If you'd like our recommendation, try out Sublime Text 3 first.

Tip: Wordpad, TextEdit, Notepad, and Word are **not** suitable code editors.

1.2 What is Python, exactly?

Ok, so python is this thing called a **programming language**. It takes text that you've written (usually referred to as **code**), turns it into instructions for your computer, and runs those instructions. We'll be learning how to write code to do cool and useful stuff. No longer will you be bound to use *others'* programs to do things with your computer - you can make your own!

Practically, Python is just another program on your computer. The first thing to learn is how to use and interact with it. There are in fact many ways to do this; the first one to learn is to interact with python's interpreter, using your **operating system's** (OS) console.

A **console** (or 'terminal', or 'command prompt') is a *textual* way to interact with your OS, just as the 'desktop', in conjunction with your mouse, is the *graphical* way to interact your system.

1.2.1 Opening a console on Mac OS X

OS X's standard console is a program called **`Terminal`**. Open Terminal by navigating to Applications, then Utilities, then double-click the **`Terminal`** program. You can also easily search for it in the system search tool in the top right.

The command line Terminal is a tool for interacting with your computer. A window will open with a command line prompt message, something like this:

```
mycomputer:~ myusername$
```

1.2.2 Opening a console on Linux

Different linux distributions (e.g Ubuntu, Fedora, Mint) may have different console programs, usually referred to as a terminal. The exact terminal you start up, and how, can depend on your distribution. On Ubuntu, you will likely want to open **`Gnome Terminal`**. It should present a prompt like this:

```
myusername@mycomputer:~$
```

1.2.3 Opening a console on Windows

Window's console is called the Command Prompt, named **`cmd`**. An easy way to get to it is by using the key combination `Windows+R` (`Windows` meaning the windows logo button), which should open a *Run* dialog. Then type **`cmd`** and hit `Enter` or click *Ok*. You can also search for it from the start menu. It should look like:

```
C:\Users\myusername>
```

Window's Command Prompt is not quite as powerful as its counterparts on Linux and OS X, so you might like to start the Python Interpreter (see just below) directly, or using the **`IDLE`** program that Python comes with. You can find these in the Start menu.

1.3 Using Python

The python program that you have installed will by default act as something called an **interpreter**. An interpreter takes text commands and runs them as you enter them - very handy for trying things out.

Just type **`python`** at your console, hit `Enter`, and you should enter Python's Interpreter.

To find out which version of python you're running, instead type `python -V` in your console to tell you.

1.3.1 Interacting With Python

After Python opens, it will show you some contextual information similar to this:

```
Python 3.5.0 (default, Sep 20 2015, 11:28:25)
[GCC 5.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note: The prompt **>>>** on the last line indicates that you are now in an interactive Python interpeter session, also called the "Python shell". **This is different from the normal terminal command prompt!**

You can now enter some code for python to run. Try:

```
print("Hello world")
```

Press `Enter` and see what happens. After showing the results, Python will bring you back to the interactive prompt, where you could enter another command:

```
>>> print("Hello world")
Hello world
>>> (1 + 4) * 2
10
```

An extremely useful command is `help()`, which enters a help functionality to explore all the stuff python lets you do, right from the interpreter. Press `q` to close the help window and return to the Python prompt.

To leave the interactive shell and go back to the console (the *system* shell), press `Ctrl-Z` and then `Enter` on Windows, or `Ctrl-D` on OS X or Linux. Alternatively, you could also run the python command `exit()`!

1.3.2 Exercise

Just above we demonstrated entering a command to figure out some math. Try some math commands of your own! What operations does python know? Get it to tell you what 239 and 588 added together, and then squared is.

1.3.3 Solution

Here are some ways you might have got the answer:

```
>>> 239 + 588
827
>>> 827 * 827
683929
```

```
>>> (239 + 588) * (239 + 588)
683929
```

```
>>> (239 + 588) ** 2
683929
```

1.3.4 Running Python files

When you have a lot of python code to run, you will want to save it into a file, so for instance, you can modify small parts of it (fix a bug) and re-run the code without having to repeatedly re-type the rest. Instead of typing commands in one-by-one you can save your code to a file and pass the file name to the **python** program. It will execute that file's code instead of launching its interactive interpreter.

Let's try that! Create a file `hello.py` in your current directory with your favorite code editor and write the print command from above. Now save that file. On Linux or OS X, you can also run `touch hello.py` to create an empty file to edit. To run this file with python, it's pretty easy:

```
$ python hello.py
```

Note: Make sure you are at your system command prompt, which will have `$` or > at the end, **not** at python's (which has >>> instead)!

On Windows you should also be able to double-click the Python file to run it.

When pressing `Enter` now, the file is executed and you see the output as before. But this time, after Python finished executing all commands from that file it exits back to the system command prompt, instead of going back to the interactive shell.

And now we are all set and can get started with turtle!

Note: Not getting "Hello world" but some crazy error about "can't open file" or "No such file or directory?" Your command line might not be running in the directory that you saved the file in. You can change the working directory of your current command line with the **cd** command, which stands for "change directory". On Windows, you might want something like:

```
> cd Desktop\Python_Exercises
```

On Linux or OS X, you might want something like:

```
$ cd Desktop/Python_Exercises
```

This changes to the directory Python_Exercises under the Desktop folder (yours might be somewhere different). If you don't know the location of the directory where you saved the file, you can simply drag the directory to the command line window. If you don't know which directory your shell is currently running in use **pwd**, which stands for "print working directory".

Warning: When playing around with turtle, avoid naming your file `turtle.py` — rather use more appropriate names such as `square.py` or `rectangle.py`. Otherwise, whenever you refer to `turtle`, Python will pick up *your* file instead of the standard Python turtle module.

CHAPTER 2

Simple drawing with turtle

2.1 Introduction

"Turtle" is a python feature like a drawing board, which lets you command a turtle to draw all over it!

You can use functions like `turtle.forward(...)` and `turtle.left(...)` which can move the turtle around.

Before you can use turtle, you have to import it. We recommend playing around with it in the interactive interpreter first, as there is an extra bit of work required to make it work from files. Just go to your terminal and type:

```
import turtle
```

Note: Not seeing anything on Mac OS? Try issuing a command like `turtle.forward(0)` and looking if a new window opened behind your command line.

Note: Do you work with Ubuntu and get the error message "No module named _tkinter"? Install the missing package with `sudo apt-get install python3-tk`

Note: While it might be tempting to just copy and paste what's written on this page into your terminal, we encourage you to type out each command. Typing gets the syntax under your fingers (building that muscle memory!) and can even help avoid strange syntax errors.

```
turtle.forward(25)
```

```
turtle.left(30)
```

The `turtle.forward(...)` function tells the turtle to move forward by the given distance. `turtle.left(...)` takes a number of degrees which you want to rotate to the left. There is also `turtle.backward(...)` and `turtle.right(...)`, too.

Note: Want to start fresh? You can type `turtle.reset()` to clear the drawing that your `turtle` has made so far. We'll go into more detail on `turtle.reset()` in just a bit.

The standard turtle is just a triangle. That's no fun! Let's make it a turtle instead with the `turtle.shape()` command:

```
turtle.shape("turtle")
```

So much cuter!

If you put the commands into a file, you might have recognized that the turtle window vanishes after the turtle finished its movement. (That is because Python exits when your turtle has finished moving. Since the turtle window belongs to Python, it goes away as well.) To prevent that, just put `turtle.exitonclick()` at the bottom of your file. Now the window stays open until you click on it:

```
import turtle

turtle.shape("turtle")

turtle.forward(25)

turtle.exitonclick()
```

Note: Python is a programming language where horizontal indenting of text is important. We'll learn all about this in the Functions chapter later on, but for now just keep in mind that stray spaces or tabs before any line of Python code

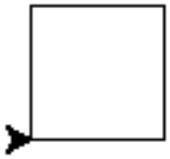
can cause an unexpected error. You could even try adding one to check how python will complain!

2.2 Drawing a square

Note: You're not always expected to know the anwer immediately. Learn by trial and error! Experiment, see what python does when you tell it different things, what gives beautiful (although sometimes unexpected) results and what gives errors. If you want to keep playing with something you learned that creates interesting results, that's OK too. Don't hesitate to try and fail and learn from it!

2.2.1 Exercise

Draw a square as in the following picture:



For a square you will probably need a right angle, which is 90 degrees.

2.2.2 Solution

```
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
```

Note: Notice how the turtle starts and finishes in the same place and facing the same direction, before and after drawing the square. This is a useful convention to follow, it makes it easier to draw multiple shapes later on.

2.2.3 Bonus

If you want to get creative, you can modify your shape with the `turtle.width(...)` and `turtle.color(...)` functions. How do you use these functions? Before you can use a function you need to know its *signature* (for example what to put between the parentheses and what those things mean.) To find this out you can type `help(turtle.color)` into the Python shell. If there is a lot of text, Python will put the help text into a *pager*, which lets you page up and down. Press the `q` key to exit the pager.

Tip: Are you seeing an error like this:

```
NameError: name 'turtle' is not defined
```

when trying to view help? In Python you have to import names before you can refer to them, so in a new Python interactive shell you'll need to `import turtle` before `help(turtle.color)` will work.

Another way to find out about functions is to browse the online documentation.

Caution: If you misdrew anything, you can tell turtle to erase its drawing board with the `turtle.reset()` directive, or undo the most recent step with `turtle.undo()`.

Tip: As you might have read in the help, you can modify the color with `turtle.color(colorstring)`. These include but are not limited to "red," "green," and "violet." See this colours manual for an extensive list.

If you want to set an RGB value, make sure to run `turtle.colormode(255)` first. Then for instance you could run `turtle.color(215, 100, 170)` to set a pink colour.

2.3 Drawing a rectangle

2.3.1 Exercise

Can you draw a rectangle too?

2.3.2 Solution

```
turtle.forward(100)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
```

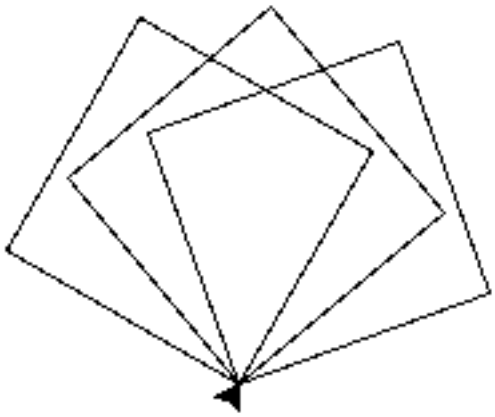
2.3.3 Bonus

How about a triangle? In an equilateral triangle (a triangle with all sides of equal length) each corner has an angle of 60 degrees.

2.4 More squares

2.4.1 Exercise

Now, draw a tilted square. And another one, and another one. You can experiment with the angles between the individual squares.



The picture shows three 20 degree turns. But you could try 20, 30 and 40 degree turns, for example.

2.4.2 Solution

```
turtle.left(20)

turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)

turtle.left(30)

turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
```

```
turtle.forward(50)
turtle.left(90)

turtle.left(40)

turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
```

CHAPTER 3

Variables

3.1 Introduction

Whew. Experimenting with the angles requires you to change three different places in the code each time. Imagine you'd want to experiment with all of the square sizes, let alone with rectangles! We can do better than that.

This is where **variables** come into play: You can tell Python that from now on, whenever you refer to a variable, you actually mean something else. That concept might be familiar from symbolic maths, where you would write: *Let x be 5.* Then *x * 2* will obviously be *10*.

In Python syntax, that very statement translates to:

```
x = 5
```

After that statement, if you do `print(x)`, it will actually output its value — 5. Well, can use that for your turtle too:

```
turtle.forward(x)
```

Variables can store all sorts of things, not just numbers. A typical other thing you want to have stored often is a **string** - a line of text. Strings are indicated with a starting and a leading `"` (double quote). You'll learn about this and other types, as those are called in Python, and what you can do with them later on.

You can even use a variable to give the turtle a name:

```
timmy = turtle
```

Now every time you type `timmy` it knows you mean `turtle`. You can still continue to use `turtle` as well:

```
timmy.forward(50)
timmy.left(90)
turtle.forward(50)
```

3.2 A variable called angle

3.2.1 Exercise

If we have a variable called `angle`, how could we use that to experiment much faster with our tilted squares program?

3.2.2 Solution

```
angle = 20

turtle.left(angle)

turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)

turtle.left(angle)
```

... and so on

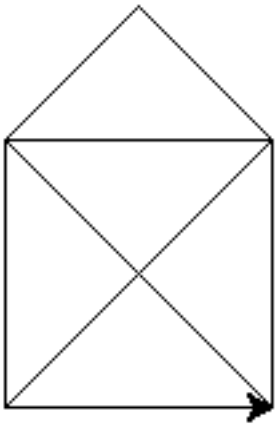
3.2.3 Bonus

Can you apply that principle to the size of the squares, too?

3.3 The house of santa claus

3.3.1 Exercise

Draw a house.



You can calculate the length of the diagonal line with the Pythagorean theorem. That value is a good candidate to store in a variable. To calculate the square root of a number in Python, you'll need to import the *math* module and use the `math.sqrt()` function. The square of a number is calculated with the `**` operator:

```
import math

c = math.sqrt(a**2 + b**2)
```

CHAPTER 4

Loops

4.1 Introduction

Something you might have noticed: our programs often feature repetition. Python has a powerful concept it makes use of called looping (jargon: *iteration*), which we can use to cut out our reptitive code! For now, **try this easy example**:

```
for name in "John", "Sam", "Jill":
    print("Hello " + name)
```

This is incredibly helpful if we want to do something multiple times — say, drawing the individual border lines of a shape — but only want to write that action once. Here's another version of a loop:

```
for i in range(10):
    print(i)
```

Notice how we write only one line of code using `i`, but it takes on 10 different values?

The `range(n)` function can be considered a shorthand for `0, 1, 2, ..., n-1`. If you want to know more about it, you can use the help in the Python shell by typing `help(range)`. Use the `q` key to exit the help again.

You can also loop over elements of your choice:

```
total = 0
for i in 5, 7, 11, 13:
    print(i)
    total = total + i

print(total)
```

Write this example out and run it with python, to check it works how you might expect.

Note: Notice how above, the lines of code that are *looped*, are the ones that are *indented*. This is an important concept in Python - that's how it knows which lines should be used in the `for` loop, and which come after, as part of the rest of your program. Use four spaces (hitting tab) to indent your code.

Sometimes you want to repeat some code a number of times, but don't care about the value of the `i` variable; so it can be good practice to replace it with _ instead. This signifies that we don't care about its value, or don't wish to use it. Here's a simple example:

```
for _ in range(10):
    print("Hello!")
```

You may or may not be wondering about the variable `i` - why is it used all the time above? Well, it simply stands for "index" and is one of the most common variable names ever found in code. But if you are looping over something other than just numbers, be sure to name it something better! For instance:

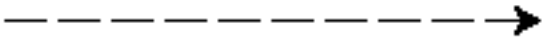
```
for drink in list_of_beverages:
    print("Would you like a " + drink + "?")
```

This is immediately clearer to understand than if we had used `i` instead of `drink`.

4.2 Drawing a dashed line

4.2.1 Exercise

Draw a dashed line. You can move the turtle without the turtle drawing its movement by using the `turtle.penup()` function; to tell it to draw again, use `turtle.pendown()`.

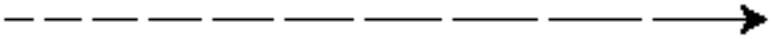


4.2.2 Solution

```
for i in range(10):
    turtle.forward(15)
    turtle.penup()
    turtle.forward(5)
    turtle.pendown()
```

4.2.3 Bonus

Can you make the dashes become larger as the line progresses?



Hint: Feeling lost? Inspect `i` at every run of the loop:

```
for i in range(10):
    print(i)
    # write more code here
```

Can you utilize `i` — commonly called the **index** variable or loop variable — to get increasing step sizes?

4.3 Comments

In the example above, the line that starts with a `#` is called a comment. In Python, anything that goes on a line after `#` is ignored by the computer. Use comments to explain what your program does, without changing the behaviour for the computer. They can also be used to easily and temporarily disable, or "comment out" some lines of code.

Comments can also go at the end of a line, like this:

```
turtle.left(20)      # tilt our next square slightly
```

4.4 More Efficient Squares

4.4.1 Exercise

The squares we were drawing at the start of this tutorial had a lot of repeated lines of code. Can you write out a square drawing program in fewer lines by utilizing loops?

4.4.2 Solution

```
for _ in range(4):
    turtle.forward(100)
    turtle.left(90)
```

4.4.3 Bonus

Try *nesting* loops, by putting one right under (*inside*) the other, with some drawing code that's inside both. Here's what it can look like:

```
for ...:
    for ...:
        # drawing code inside the inner loop goes here
        ...
    # you can put some code here to move
    # around after!
    ...
```

Replace the `...`'s with your own code, and see if you can come up with something funny or interesting!
Mistakes are encouraged!

CHAPTER 5

User-defined functions

5.1 Introduction

Programmers can deal with some pretty complex and abstract problems, but one sign of a good programmer is that they're lazy. They only like to deal with one thing at a time. So you need a way to break up problems into smaller, discrete pieces, which lets you focus on just the piece you want to.

Functions are one way to do this abstraction in Python. Let's take `turtle.reset()` for example. `reset` is a function we call on our `turtle`, and it is actually an abstraction for a number of steps, namely:

- Erase the drawing board.
- Set the width and color back to default.
- Move the turtle back to its initial position.

But because all the code is contained in the function, we don't have to worry about these details. We can simply *call* this function, and know it will do what it says for us.

So - how to write your own?

A function can be defined with the `def` keyword in Python:

```
def line_without_moving():
    turtle.forward(50)
    turtle.backward(50)
```

This function we defined is called `line_without_moving` and it is an abstraction for two turtle steps - a move forward and a move backward.

To use it (or as it is usually called, "to call it"), write its name followed by parentheses:

```
line_without_moving()
turtle.right(90)
line_without_moving()
turtle.right(90)
line_without_moving()
turtle.right(90)
line_without_moving()
```

We could write more functions to remove some of the repetition:

```
def star_arm():
    line_without_moving()
    turtle.right(360 / 5)
```

```
for _ in range(5):
    star_arm()
```

Important: Python uses *indenting with whitespace* to identify blocks of code that belong together. In Python a block (like the function definitions shown above) is introduced with a colon at the end of the line and subsequent commands are indented — usually 4 spaces further in. The block ends with the first line that isn't indented.

This is different to many other programming languages, which use special characters (like curly braces `{}`) to group blocks of code together.

Never use tab characters to indent your blocks, only spaces. You can – and should – configure your editor to put 4 spaces when you press the tab key. The problem with using tab characters is that other python programmers use spaces, and if both are used in the same file python will read it wrong (in the best place, it will complain, and in the worst case, weird, hard to debug bugs will happen).

5.2 A function for a square

5.2.1 Exercise

Write a function that draws a square. Could you use this function to improve the tilted squares program? If you change the program to use a function, is it easier to experiment with?

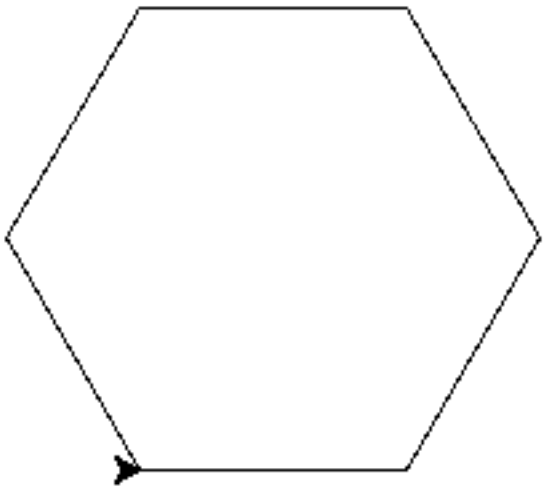
5.2.2 Solution

```
def tilted_square():
  turtle.left(20)      # now we can change the angle only here
  for _ in range(4):
      turtle.forward(50)
      turtle.left(90)

tilted_square()
tilted_square()
tilted_square()

# bonus: you could have a separate function for drawing a square,
# which might be useful later:

def square():
  for _ in range(4):
      turtle.forward(50)
      turtle.left(90)

def tilted_square():
  turtle.left(20)
  square()

# etc
```

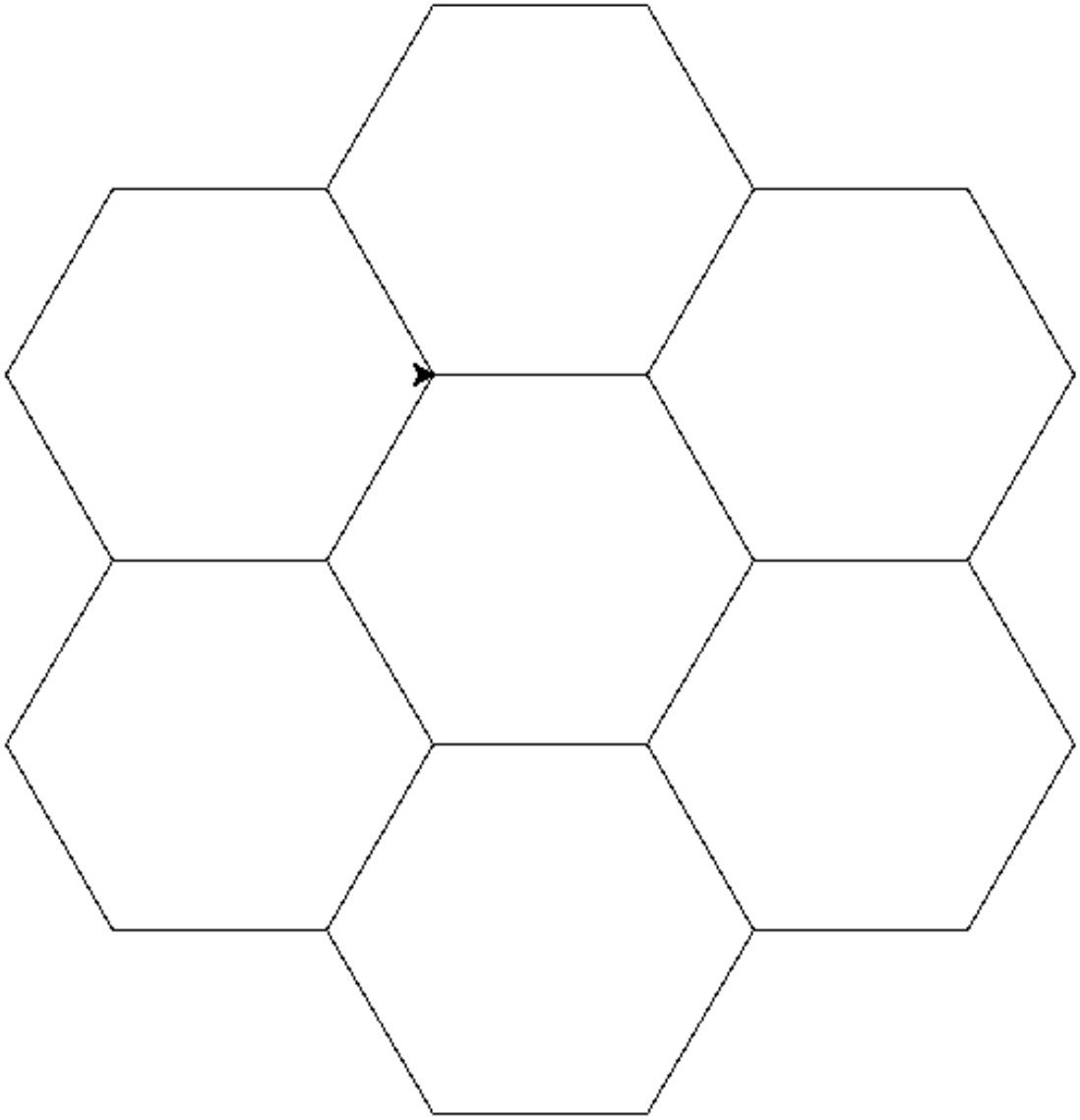
5.3 A function for a hexagon

5.3.1 Exercise

Write a function that draws a hexagon.



Now combine that function into a honeycomb. Just make it with a single layer like this:



Give it a good go!

Hint: Make sure your hexagon function returns your turtle to exactly the same position and angle it was before it was asked to draw the hexagon. This makes it easier to reason about.

5.3.2 Solution

```
def hexagon():
  for _ in range(6):
      turtle.forward(100)
      turtle.left(60)

for _ in range (6):
    hexagon()
```

```
    turtle.forward(100)
    turtle.right(60)
```

You could also put the `turtle.forward(100); turtle.right(60)` portion in the function, but you better not call it `hexagon` in that case. That's misleading because it actually draws a hexagon and then advances to a position where another hexagon would make sense in order to draw a honeycomb. If you ever wanted to reuse your hexagon function outside of honeycombs, that would be confusing.

CHAPTER 6

Functions with parameters

6.1 Introduction

As we shrink down our code and add functions to remove duplication, we are *factorizing* it. This is a good thing to do. But the functions we have defined so far are not very flexible. The variables are defined inside the function, so if we want to use a different angle or a distance then we need to write a new function. Our hexagon function can only draw one size of hexagon!

That is why we need to be able to give parameters, also called *arguments*, to the function. This way the *variables* in the function can have different values each time we call the function:

Remember how we defined the function `line_without_moving()` in the previous section:

```
def line_without_moving():
    turtle.forward(50)
    turtle.backward(50)
```

We can improve it by giving it a parameter:

```
def line_without_moving(length):
    turtle.forward(length)
    turtle.backward(length)
```

The parameter acts as a *variable* only known inside the function's definition. We use the newly defined function by calling it with the value we want the parameter to have like this:

```
line_without_moving(50)
line_without_moving(40)
```

We have been using functions with parameters since the beginning of the tutorial with `turtle.forward()`, `turtle.left()`, etc...

And we can put as many arguments (or parameters) as we want, separating them with commas and giving them different names:

```
def tilted_line_without_moving(length, angle):
    turtle.left(angle)
    turtle.forward(length)
    turtle.backward(length)
```

6.2 A parameterized function for a variable size hexagon

6.2.1 Exercise

Write a function that allows you to draw hexagons of any size you want, each time you call the function.

6.2.2 Solution

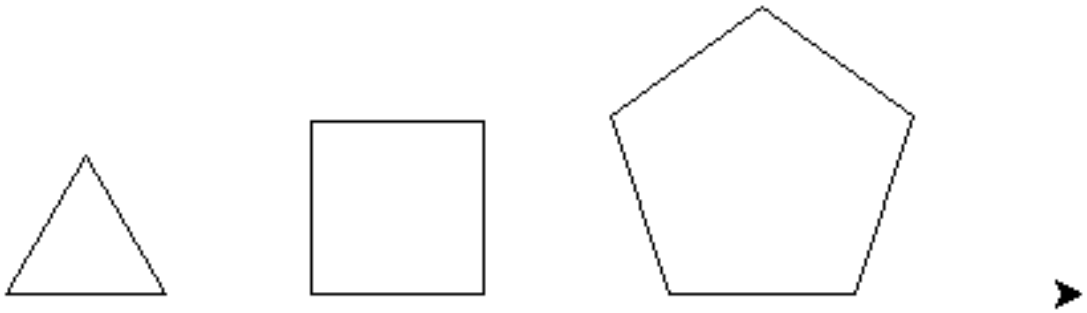
```
def hexagon(size):
    for _ in range(6):
        turtle.forward(size)
        turtle.left(60)
```

6.3 A function of several parameters

6.3.1 Exercise

Write a function that will draw a shape of *any* number of sides (let's assume greater than two) of any side length. Get it to draw some different shapes.

Here's an example of drawing shapes with this function:



Tip: The sum of the external angles of any shape is always 360 degrees!

6.3.2 Solution

```
def draw_shape(sides, length):
    for _ in range(sides):
        turtle.forward(length)
        turtle.right(360 / sides)
```

6.3.3 Bonus

It might sound crazy, but it's perfectly possible to pass a *function* as a parameter to another function! Python regards functions as perfectly normal 'things', the same as variables, numbers and strings.

For instance, you could create a shape drawing function which turned one way or another depending on which turtle function you passed to it - `turtle.left` or `turtle.right`.

See if you can implement this!

Note: Passing a function (e.g `turtle.left`) is different than *calling* it, which would instead be written `turtle.left(45)`.

CHAPTER 7

Conditional statements

7.1 Introduction

So far we have accomplished predefined tasks, but in all honesty we were accomplishing no better than old-time music boxes following one set of instructions to the end. What makes programming so much more powerful are conditional statements. This is the ability to *test* a variable against a value and act in one way if the condition is met by the variable or another way if not. They are also commonly called by programmers *if statements*.

To know if a condition is *True* of *False*, we need a new type of data: the booleans. They allow logical operations. A logic statement or operation can be evaluated to be *True* or *False*. Our conditional statement can then be understood like this:

> **if *(a condition evaluates to True)*:** *then do these things only for 'True'*
>
> **else:** *otherwise do these things only for 'False'.*

The condition can be anything that evaluates as *True* or *False*. Comparisons always return *True* or *False*, for example == (equal to), > (greater than), < (less than.)

The **else** part is optional. If you leave it off, nothing will happen if the conditional evaluates to 'False'.

7.2 Examples

Here are some examples. You may want to read them over line-by-line and see what you think they do, or run them to be certain:

```
condition = True
if condition:
    print("condition met")

if not condition:
    print("condition not met")

direction = -30
if direction > 0 :
    turtle.forward(direction)
else:
    turtle.left(180)
    turtle.forward(-direction)
```

7.3 Giving Directions

Python turtles can be very good at following instructions. Let's use the `input()` function to ask the user for a direction to move the turtle. To keep things easy we will only accept two instructions: "left" and "right".

Note: Using Python 2? The `input()` function is called `raw_input()`.

It's much easier to define this as a function, like so:

```
def move():
    direction = input("Go left or right? ")
    if direction == "left":
        turtle.left(60)
        turtle.forward(50)
    if direction == "right":
        turtle.right(60)
        turtle.forward(50)
```

Now whenever you use `move()` you are prompted to choose `left` or `right`.

7.4 "data munging"

In this program, the turtle will only respond to exactly `left` or `right` with no variation. Though `Left` or `LEFT` might seem the same as `left` to a human, that isn't the case when programming. Python has a few utility methods to help with that. A string has the methods `.strip()`, which removes whitespace and `.lower()` which makes everything lower-case.

Here are some examples to print out the effects of `.strip()` and `.lower()`:

```
my_variable = "       I Am Capitalised"
print(my_variable)
my_stripped = my_variable.strip()
print(my_stripped)
my_lower = my_variable.lower()
print(my_lower)
```

Try adding `direction = direction.strip().lower()` to the `move()` function, to see the effect. We often call this kind of code "data munging", and it is very common.

Can you add some extra input choices to make the turtle draw other things? How about `hexagon`?

CHAPTER 8

Conditional Loops

8.1 Introduction

Conditional loops are way to repeat something while a certain condition is satisfied, or **True**. If the condition is always satisfied (never becomes **False**), the loop can become infinite. If the condition starts off false, the code in the loop will never run! In Python conditional loops are defined with the `while` statement:

```
word = ''
sentence = ''
print('Please enter some words.')
print('Include a period (.) when you are finished.')
while '.' not in word:
    word = input('next word: ')
    sentence = word + ' ' + sentence
print()
print('Aha! You said:')
print(sentence)
```

We call this part of the code the 'conditional': `'.' not in word`

Whether the conditional returns True or not determines whether the code inside the `while` loop runs. Of course, it repeats the check every time the loop is run again.

Read the code above, and see if you can summarise in your head what it should do (what its final output will be).

Then copy it into a file, say `sentence.py` and run it - see exactly what it does. Does that match up with what you thought?

Note: If you are using Python 2, you will need to replace `input` with `raw_input` to run the program correctly.

8.2 Turtle prison

8.2.1 Exercise

The turtle has been up to its usual tricks again, robbing liquor stores and building up huge gambling debts. It's time for turtle to be put into a cell that it can't get out of.

Let's make a new version of `forward()`. One that will turn the turtle around if it tries to go further than 100 from the origin. We'll need a `while` loop, and some new turtle functions:

- `turtle.distance(0, 0)` - Returns the distance of the turtle from the origin (0, 0)
- `turtle.towards(0, 0)` - Returns the angle to get back to origin (0, 0)
- `turtle.setheading(angle)` - Directly sets the turtle's direction

You could try playing with a turtle in the interpreter and using these functions to check exactly what they do, if you like.

Now you will need to implement the prison logic using these turtle functions, perhaps a `while` loop and a bit of conditional logic. It's a bit of a stretch but keep at it! Don't be afraid to talk it out with a coach or another student.

8.2.2 Solution

```
def forward(distance):
    while distance > 0:
        if turtle.distance(0,0) > 100:
            angle = turtle.towards(0,0)
            turtle.setheading(angle)
        turtle.forward(1)
        distance = distance - 1
```

8.3 Draw a spiral

Loops can be interrupted with the `break` statement. This is especially useful if you write an *infinite loop*, which is a loop where the conditional is always **True**.

8.3.1 Exercise

Write a `while` loop with a condition that is always **True** to draw a spiral. Interrupt the loop when the turtle reaches a certain distance from the center. Use the function `turtle.distance(x, y)` to get the turtle's distance to the point defined by the coordinates `x` and `y`.

To do this you will need the `turtle.xcor()` and `turtle.ycor()` functions, which return the position of the turtle in X and Y axes respectively.

Note: To draw a spiral, the turtle has to rotate by a constant value and move forward by an increasing value.

8.3.2 Solution

```
def draw_spiral(radius):
    original_xcor = turtle.xcor()
    original_ycor = turtle.ycor()
    speed = 1
    while True:
        turtle.forward(speed)
        turtle.left(10)
        speed += 0.1
        if turtle.distance(original_xcor, original_ycor) > radius:
            break
```

8.3.3 Bonus

Can you make a conditional for this loop, so you don't need the infinite loop `while True` or the `break`? Which version do you find easier to understand?

CHAPTER 9

Logical operators

9.1 Introduction

Conditionals are a nice way to make decisions by asking if something equals *True* or not. But often one condition is not enough. We may want to take the opposite of our result. Or for instance if we want to make a decision upon `turtle.xcor()` and `turtle.ycor()` we have to combine them. This can be done with logical operators.

9.2 Negation of a statement

If we want something to be *False* we can use `not`. It is a logical operator:

```
x = False
if not x :
    print("condition met")
else:
    print("condition not met")
```

9.2.1 Exercise

The turtle gives us a useful function to know if it is drawing or not: `turtle.isdown()`. This function returns *True* if the turtle is drawing. As we have seen earlier, the function `turtle.penup()` and `turtle.pendown()` toggle between drawing while moving, or just moving without a trace.

Can we write a function that only goes forward if the pen is up?

9.2.2 Solution

```
def stealthed_forward(distance):
    if not turtle.isdown():
        turtle.forward(distance)
```

9.3 This and that or something else

Two easy to understand operators are `and` and `or`. They do exactly what they sound like::

```
if 1 < 2 and 4 > 2:
    print("condition met")

if 1 > 2 and 4 < 10:
    print("condition not met")

if 4 < 10 or 1 < 2:
    print("condition met")
```

You are not restricted to one logical operator. You can combine as may as you want.

9.3.1 Exercise

Earlier we put the turtle in a circular prison. This time let's make it a box. If the turtle goes more than 100 in the X *or* Y axis then we turn the turtle back around to the center.

9.3.2 Solution

```
def forward(distance):
    while distance > 0:
        if (turtle.xcor() > 100
            or turtle.xcor() < -100
            or turtle.ycor() > 100
            or turtle.ycor() < -100):
            turtle.setheading(turtle.towards(0,0))
        turtle.forward(1)
        distance = distance - 1
```

CHAPTER 10

Where to go from here

10.1 Learning Python

Hopefully this tutorial has tought you just enough python to get you on your feet. However, there is much more that you can learn! Even professional programmers will always be trying to learn more about their language and how to code excellently with it.

10.1.1 Books and Tutorials

If you like learning from a book, there are heaps of good ones, and even many that can be read freely on the web!

- OTS Python Portal - Check the rest of our tutorials, and look out for future workshops!
- O'Reilly publishes hundreds of books and ebooks, on python and many other technologies (check out Learning Python, 5th edition).
- Invent with Python takes a practical approach, with three different ebooks that can be read online for free.
- Learn Python the Hard Way is a step by step full tutorial on the language, done in a unique style.
- Dive into Python 3 is another great book available online for free!
- Last but not least, python.org has Its own tutorial!

10.1.2 Online learning courses

You wouldn't believe what you can do on the web these days. There are many courses on programming around!

- Edx - with top universities like Stanford, MIT, Harvard and Berkley giving out interactive courses for free, it's hard to turn this down. You'll have to look for ones coming up soon!
- Coursera is very similar to Edx, with even more courses, and some that can be taken at any time.
- CodeAcademy - luckily this one has a python-specific course that can be taken at any time, and many other practical language courses.

10.2 What to do with Python

Well, that's a tough one! See, practically anything you can think of involving some electronics - from your TV remote, to a smartphone, to the backend of a popular web service, to the scheduling of airport landings, to the software you use everyday - it all involves some programming somewhere!

And not only is it a big field practically, but also technically and academically. As soon as you have a basic understanding of a general language like python, you can start working on all manner of subjects: web frontends, web backend services, data analysis and statistics, Artificial Intelligence, GUI design, robotics, software development of all kinds, online transactions, automation of everything, and many more.

We wouldn't blame if that all sounds daunting, so here's some practical advice, that takes advantage of how wide a field "programming" is:

Apply your new skills to whatever you're passionate about.

Whether that's starting a new business, helping you do some task involving data faster, putting up an online site, calculating advantages in a game you play, creating art and music, or anything else, finding out how to use programming to help with whatever floats your boat will make learning much more exciting and relevant. Feel free to ask someone more experienced if you have no idea where to start, but always mention what makes you tick!

CHAPTER 11

License

11.1 Contributors

The following people have contributed to this material, in alphabetical order:

- Alec Clews
- Alper Çugun
- Amélie Anglade
- Andreas Hug
- Angus Gratton
- Aur Saraf
- Benjamin Kampmann
- Benoît Bleuzé
- Charles Pletcher
- Haiko Schol
- julius.jann
- leenagupte
- Markus Zapke-Gründemann
- Matt Iversen
- Matthew Iversen
- OKso
- Robert Lehmann
- Robert Schwarz
- sorrymak
- staeff
- Steven Farlie

(This list is automatically generated from our source repository.)

The material is legally maintained by

OpenTechSchool e.V.
c/o co.up
Adalbertstr. 8

10999 Berlin, Germany