Introduction to Programming with Python Documentation

Release 2015.03.28

OpenTechSchool

March 28, 2015

Contents

Table of Contents:

CHAPTER 1

Introduction

Learning to program requires much practice. However it is also very rewarding and creative.

The purpose of this course it to give you enough programming vocabulary to get a taste of what it is.

1.1 print('Hello')

1. Type *Windows* + *R* (the two keys together). A search box pops up.
2. Type *cmd.exe* and press enter.

The program *cmd.exe* will launche and you should see a prompt:

```
C:\Users\greg-lo>
```

The line tells you your current location followed by >.

cmd.exe is a program often called a shell. It is an alternative to the point and click that we are all used to.

Typing *python3* and you enter the python interpreter:

```
C:\Users\greg-lo> python3
Python 3.4.2 (v3.4.2:8711a0951384, Sep 21 2014, 21:16:45) [MSC v.1600 32 b
it (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note the prompt has changed to >>>. Python is waiting for your instructions.

Tip: Do not confuse the shell with the python interpreter.

If the prompt is > you are in cmd.exe If the prompt is >>> you are in the python interpreter.

In the interpreter type the following and press enter:

```
>>> print('Hello')
```

Congratulations you have just written and executed a line of python!

1.1.1 Exercise

1. Explore and experiment with the interpreter. Try printing other words.

2. Can you make Errors appear (it shouldn't be too difficult)? How many different ones can you make? Make a list and google each one.

1.2 Errors

Troubleshooting errors is a large part of programming.

Typically given a problem to solve a programmer thinks up an idea that may work then battles through errors until it does work.

Often beginners understandably get frustrated with them. Don't. Instead build up resilience by taking time to understand them. They are always correct and trying to guide you to a solution.

Here are three you will see a lot:

```
>>> def asdfwe:
  File "<stdin>", line 1
    def asdfwe:
              ^
SyntaxError: invalid syntax

>>> if 5 > 6:
... print('yes')
  File "<stdin>", line 2
    print('yes')
        ^
IndentationError: expected an indented block

>>> def asdfwe:
  File "<stdin>", line 1
    def asdfwe:
              ^
SyntaxError: invalid syntax
```

By the end of this course, you should be able to instantly map the above errors to solutions.

Tip: Troubleshooting Errors:

1. Reading error messages. Try to intuitively solve them.
2. Google errors. There isn't a single error someone hasn't already had.
3. Ask an expert. If really stuck ask someone for help.

1.3 Object oriented programming

We can see our world as containing different types of objects that we can classify according to common attributes and behaviours.

For example in a classroom there many objects that are instances of the type Chair and many other objects that are instances of the type Person.

Objects have:

- Attributes - Chairs have four legs, Persons have two.
- Behaviours - Persons can walk. Persons can move chairs.

Objects can interact with other objects of different types. An object of type Person can can sit on an object of type Chair.

This is the essence of object oriented thinking. It is about using programmatic objects to model a domain of interest to a programmer.

This style of programming provides a clear, simple, and consistent model of computation that maps well to our intuitions about the world.

1.4 Python

Python is a simple to learn yet fully featured, high-level, object oriented programming language. It's popular both in academia, science and other industries. The concepts however will apply to most other object oriented languages.

Two things to bear in mind:

- A language - The textual instructions you type.
- An interpreter - A program (called *python*) that reads and executes that language.

Together we will learn the correct syntax and grammar of the Python language.

When we ask the interpreter to execute it, it is interpreter that understands how to translate Python scripts into creating and manipulating objects according to your instructions.

This course introduces different types of Python objects: *String*, *Integer*, *Turtle*, *lists*...

You will discover what attributes and behaviours these objects have and how to use these to write programs to get stuff done.

1.4.1 Questions

1. What other languages have you heard of?
2. Explain in your own words but using the concepts *interpreter* and *language* what happened above when you printed text.

1.5 Language goals

Our goal is move from this:

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

to this:

```
def square(side):
    for i in range(4):
        turtle.forward(side)
        turtle.left(90)
```

1.5.1 Questions

Amongst yourselves:

- What does the first code extract do?
- What does the second code extract do?
- Which do you prefer and why?

CHAPTER 2

Turtles

Turtle objects know how to draw. Here we explore creating and manipulating them to draw on the screen.

We also look at the two ways the *python3* interpreter can execute your python code:

- The interactive interpreter
- Calling the python interpreter on a file that contains code.

Tip: Don't just read! Type everything and experiment.

2.1 Interactive interpreter

We launch the python interpreter through *cmd.exe*:

1. Press the two keys 'Windows + R' together
2. Enter *cmd.exe* in the search prompt and enter.

A window will appear with a prompt:

```
C:\Users\greg-lo>
```

Type *python3* to enter the interactive shell:

```
C:\Users\greg-lo>python
Python 3.4.2 (v3.4.2:8711a0951384, Sep 21 2014, 21:16:45) [MSC v.1600 32 b
it (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> print('Hi')
```

Now type:

```
>>> from turtle import Turtle
>>> tess = Turtle()
```

```
>>> tess.forward(100)
```

```
>>> tess.left(30)
```

Lets call some more methods on the tess our turtle object:

```
>>> tess.shape('turtle')
>>> tess.color('green')
```

Lets create 'bob' a new turtle object:

```
>>> bob = Turtle()
>>> bob.shape('circle')
>>> bob.color('red')
>>> bob.backward(100)
```

2.1.1 Exercise

Experiment drawing shapes in different colours.

2.1.2 Documentation

Visit the *turtle* online documentation and explore what Turtle objects can do.

Questions:

- What different colors does a turtle's *color* method recognise?
- What shapes does a trutle's *shape* method recognise?

Find some new turtle object methods and experiment.

Tip: As you experiment you will want to do know how to do new things. Get into the habit of exploring the documenation to see what you can do.

2.2 Turtles

Lets revise what we have learnt in the light of object oriented terminology.

An object can be created. It has a type, and this type determines its methods (behaviours).

2.2.1 Creation

```
>>> from turtle import Turtle
>>> tess = Turtle()
```

Breakdown:

1. We import an object called Turtle from somewhere called turtle.
2. Turtle is called, creates a new object of type turtle, and returns it.
3. This returned object is assigned to the name tess.

Tip: We call an object by adding parenthesis at the end of its name. Here the parenthesis are empty but then often aren't.

Lets confirm the type of tess:

```
>>> type(tess)
turtle.Turtle
```

Tip: The function *type* returns the type of a passed object.

Turtle is a special kind of object in that it produces new objects. We call it a constructor object.

2.2.2 Methods

Methods are functions attached to objects. We will explore functions later.

```
>>> tess.forward(100)
```

Braces *()* have a special meaning. They indicate calling. You can think of this as effecting an action.

The effect of calling the method *forward* on an object of type *Turtle* is to draw a line.

What other methods (behaviours) do turtle objects have?

2.3 Code in files

Most code is written and executed from a file.

Using SublimeText create a file named *my_turtle_file.py* with this code:

```
from turtle import Turtle, exitonclick

tess = Turtle()
tess.shape("turtle")
tess.forward(100)

exitonclick()  # Why this? Experiment by commenting it out.
```

Tip: All word document file names end with .doc, all files names with python code must end with .py

In cmd.exe call the python command with the filename *my_turtle_file.py* as parameter:

```
C:\Users\greg-lo> python3 my_turtle_file.py
```

Tip: Make sure the file you created exists in the location where you execute this command. The location is given by the prompt.

2.3.1 Questions/Practicals

1. What are the differences between using *python3* interactively and using files? When would you use one or the other?
2. Challenge yourself to find as many different ways of drawing with a turtle object.
3. Take your time to draw something useful and/or crazy.

2.4 Shape Exercises

Lets program some shapes. We do this by breaking down into step by step instructions principles of geometry.

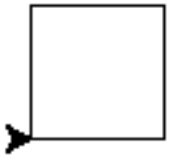
Put all code inside a file named *shapes.py* to be executed using:

```
python shapes.py
```

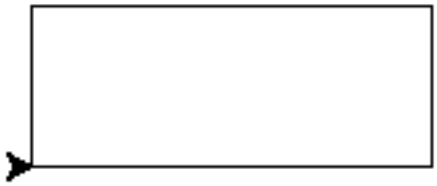
Shapes:

- Draw a square as in the following picture.

Tip: Squares have right angles which are 90 degrees.



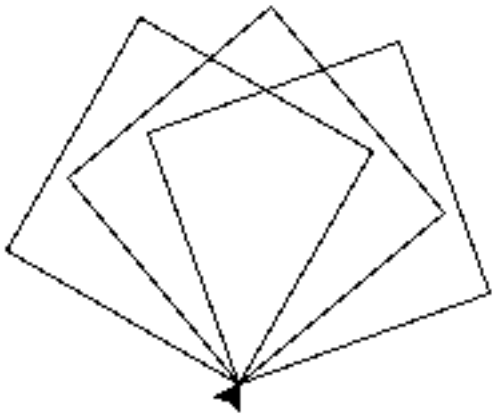
- Draw a rectangle.



- Draw an equilateral triangle.

Tip: An equilateral triangle has 3 sides of equal length and each corner has an angle of 60 degrees.

- Draw many squares. Each square should be tilted left of the previous.



Experiment with the angles between the individual squares. The picture shows three 20 degree turns. You could try 30 and 40...

- Draw a simple house.

Tip: Reuse the code you have already written.

CHAPTER 3

Numbers

In this and the following section we examine three new types of objects

Type	Used for	Examples
Integers	Numbers	**-5** 16
Floats	Decimal	3.4 -23.001
Strings	Text	'abc' 'bob'

We will look at how to create objects of these types and what operators (behaviours) they respond to.

3.1 Integers & Floats

int objects represent natural numbers. *float* objects represent rational numbers, numbers that have a decimal value. Both types can represent positive or negative numbers.

3.1.1 Creation

Unlike creating turtles there is no formality to creating an *int* and *float* objects.

You just type them as literals:

```
>>> 3
3
>>> type(-5)        # confirm type
int
>>> 3.4
3.4
>>> type(3.0)
<class 'float'>
```

3.1.2 Questions

1. Why do we have two different types to represent numbers?
2. Find some uses cases where you'd choose an *int* and others where a *float* is more suitable.

3.2 Number Operators

Tip: Unlike turtle object methods, we use operators to manipulate number objects. This special syntax exists as it maps to our expectations and so is more intuitive.

3.2.1 Arithemtic operators

Two number objects separated by an arithmetic operator */ - +, actions behaviour we expect from basic arithmentic.

```
>>> 5 + 4
9
>>> 5 - 6
-1
```

The behaviour is to compute and return the result as a new object with the same type.

3.2.2 Comparison operators

Likewise two number objects separated by comparison operators == != >= <= < >, have the behaviour we expect.

```
>>> 5 == 4
False                # What is this?
>>> 5 < 6
True                 # and this?
>>> 6 <= 6
True
```

Tip: *int* objects are used to solve problems that require manipulating numbers but with no decimal point such as age, and days, IDs.

These are expressions and these evaluate to *True* or *False*.

3.3 The *if* conditional

This pattern:

```
if <boolean expression>:
    <block of code>            # Note 4 space indent
```

mirrors the syntax required to define conditional behaviour.

Typically we use the result of comparison statements to make decisions on what code to execute.:

```
if 6 > 5:
    print('Greater Than')
```

if statements can combine with else:

```
if 6 > 5:
    print('Greater Than')
else:
    print('Not Greater Than')
```

3.3.1 Questions

Find some uses cases where you'd use the if conditional.

3.4 Number Exercises

1. A bar wants to ensure only adults are allowed in. Write a program in a file named *bar.py* that prints 'underaged' or 'ok' depending on the age entered in the code.
2. A ride operator needs to ensure clients are taller than 150cm due to security. Write a prgram in a file named *ride.py* that will print 'ok' or 'not tall enough' given a height entered in the code.
3. A trader wants to algorithmically buy 'ACME' corp stock if they rise above 0.005$ but sell if they are below 0.001$. Write a script *trader.py* that prints 'buy', 'sell', 'hold' depending on a sale price entered in the script.

CHAPTER 4

Text

4.1 Strings

Text is reprsented and manipulated using objects instances of type *str*.

4.1.1 Creation

```
>>> "hi"
hi
>>> type('hi')      # confirm type
str
```

When you execute the code *"hi"* or *str("hi")*, the python interpreter:

1. Creates an object of type *str*
2. Gives it the value "hi"
3. Returns this newly created object

4.1.2 Methods

Many! Consult the online documentation:

4.1.3 Questions

1. Find practical uses cases of where you'd like to manipulate text.
2. Using the above documentation and the interpreter interactively try to map those use cases to actual code solutions.
3. Try to imagine use cases for each of the methods that exist on *str* objects in the docs.

4.1.4 Conversion functions

Often we need to convert between numbers and text. (Why?)

You can use the *int* and *float* functions to convert *str* objects into number objects:

```
>>> int('3')
3
>>> float('3.4')      # constructor can convert from str
3.4
>>> str(3)
'3'                   # note the ''s that indicate a str object
>>> str(3.4)
'3.4'
```

4.2 User Input

To make programs interactive use a function named *input*:

```
>>> name = input("Please enter your name: ")
Please enter your name:
```

When the interpreter meets *input* it:

1. prints the string message passed as an argument to *input*,
2. Buffers (stores) any characters typed
3. On *enter* returns the characters as a new String.

Here the resultant string is assigned to the name *name*.

So if the user types in *Sophocles* then enter, a string object of value 'Sophocles' is assinged to name.

4.3 *if* and *elif*

We can define more complex conditional behaviour by combining if with elif and else:

```
>>> x = input("Enter your age: ")     # input returns a str
Enter your age: 24
>>> x = int(x)                        # convert to an int
>>> if x < 18:
...     print('You are a child')
... elif x == 18:
...     print('You have just turned into an adult')
... else:
...     print('You are an adult')
```

4.4 Exercises

1. Rewrite the number programs *bar.py*, *ride.py* and *trader.py* to take input from the user. Think of an appropriate question to print to screen to solicit a correct response.

What if the user enters nonsense? There is rarely a program without some form of validation. This is explored in the next exercise.

2. A sign up form on a website for the company 'Very Big Corp. Of America' requires information from its clients. The company wants to do gender based email marketing. Put this code in *big_corp.py*

a Write a program that asks clients their name, address, and gender. Ensure that gender is represented as either 'm', 'f'. If it is not ask the user again. Once all information is inputted print 'Hi Mr Greg, we have shaving blades reduced this week' and for women 'Hi Ms Natalia, we have cosmetics currently on sale'

b The same program now requires people to enter their email address. Add this but ensure it is well formed. You will need to define what a well formed email address is.

3. A mobile phone company bills clients on a certain plan differently depending on whether they have dialed a number containing 0845 or not. Write a program that asks the user which number they'd like to dial and answers whether it is 'free' or 'paid'. Use *mobile.py*

4. A geneticist needs your help identifying if a dna sequence exists in a larger strand of dna. A DNA sequence consists of a sequence of A, T, G, and Cs. Write a program that takes a DNA sequence from the user and confirms 'Found' or 'Not Found' depending on whether the input is contained in the target DNA strand. Use *dna.py*

   DNA strand: ATTGCGCCTTATGCTTAACC

   As a challenge extend this program to check that the input is correct.

CHAPTER 5

Names

In this section we will examine names and assignment more closely.

Tip: Names and variables in this context are synonyms. We use 'name' because Python uses this terminology.

5.1 Assignment

Assignment is one of the ways we associate names with objects. Names are how the interpreter knows to locate what programmers are referring to.

Just like we need to name person objects to know how to differentiate between them, *python3*, the interpreter, also needs to know what you are referring to when you give it instructions.

Tip: = the equals symbol means assignment and not equality (unlike in maths).

```
>>> x = 5
```

The interpreter executes the above code as:

1. Create an *int* object of value 5.
2. Does *x* exist in the namespace?

   True - update the name *x* to point to the new *int* object.

   False - create a new name *x* in the namespace and make it point to the new object.

From the point of assignment onwards code can refer to that object by using the name *x*. The interpreter will know how to find it by looking up the value in the namespace.

A name is an expression and it evaluates to its object:

```
>>> x
5
```

Names can be reassigned to any type of object:

```
>>> x = 5             # x refers to an 'int' object
>>> x = 'greg'        # x refers now to a 'str' object
```

The mysterious *from ... import ...* that we saw earlier is just about adding names to the namespace so the interpreter knows what you are referring to:

```
>>> from turtle import Turtle
>>> tess = Turtle()
```

5.1.1 Visualising

For each assignment:

- If the name already exists, the namespace (frame) is updated.
- If the name doesn't exist, a new name is created pointing and it references the newly created object.

5.1.2 NameError

If the interpreter gets a name that hasn't yet been defined through assignment it will complain by throwing a *NameError*.

example:

```
>>> the_holy_grail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'the_holy_grail' is not defined
```

5.1.3 Questions

```
five = "five"
```

What does each set of characters on either side of the equal sign mean?

5.2 Reusability

Names enhance a programmers' expressivity. They permit generalising code thereby facilitating code reuse. Indeed they are often called variables.

Consider this code that draws a square with side length 50:

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

Now a decision is made that the sides be of length 100.

You have to go back and replace 50 with 100 four times.

Using names you can do this:

```
side = 50
right_angle = 90

turtle.forward(side)
turtle.left(right_angle)
turtle.forward(side)
turtle.left(right_angle)
turtle.forward(side)
turtle.left(right_angle)
turtle.forward(side)
turtle.left(right_angle)
```

If you change your mind you need only update one value.

Mathematics tells us a square's length can be of any size. Our new programmatic definition mirrors that.

Tip: If you find yourself needing to replace many similar values in order to update your code, using names is worth considering.

5.2.1 Good Naming

The name *right_angle* was chosen to refer to an *int* of value 90.

We could have used *thirty_degree_angle*, *angle*, or *awef* and the code would work fine. However:

- *thirty_degree_angle* is misleading its 90 not 30 degrees.
- *angle* is perhaps ok but a little vague
- *awef* is nonsense and conveys no meaning

By choosing appropriate names you make the code more readable and intuitive.

5.3 Exercises

5.3.1 Age in 2050

Write a program that asks the user for her age and prints how old she will be in 2020.

5.3.2 Shapes

Refactor your code in *shapes.py* to use variables as much as possible.

5.4 Objects & Types Q&A

If you understand the answers to these you understand everything about objects and types!!

Tip: Use the interpreter to help you find answers

Describe in detail what the interpreter does when you type the following and enter:

```
>>> '5'
```

```
>>> 5
```

What is the result this line of code?:

```
3 < '5'
```

Instances of both *str* and *int* objects recognise the + symbol. What output would you expect of the following lines of code?

```
'1' + '2'
```

```
1 + 2
```

Try the same above but this time using * instead of +. What can you conclude of the meaning of *?

CHAPTER 6

Conditionals

Conditional flow control is how the python interpreter chooses which code to execute. Think of it as how to express choices.

Boolean expressions are lines of code that resolve to a boolean object. There are only two values a boolean object can take: True or False.

Conditionals always base their decisions on the result of a boolean expression. They are always followed by a block of code.

Furthermore conditional loops enable us to harness logic relating to repetition.

6.1 Code Blocks

A block of code is code that will execute together. A block is defined by the use of indentation.

All types of conditionals use code blocks which are executed depending on the outcome of the conditional expression that guards their execution.

```
a = 4
if a == 4:
    print('This code block will execute')
    result = 5 + a
else:
    print('This code block will not execute')
    result = a + 6
```

Tip: In other languages code blocks are defined by the use of braces '{}'s

6.2 Equality

Testing the equality of two objects returns *True* or *False* depending on how equality is defined on those two objects.

Equality on stings is defined as follows:

```
>>> '5' == '5'
True
>>> '5' == '6'
False
```

Generally the objects have to first be of the same type and then have the same value to be equal:

```
>>> 5 == '5'
False
```

6.3 The *while* loop

The *while <condition>:* construct is a way of instructing the interpreter to repeat indefinitely. The condition defines when the loop will terminate.

6.3.1 syntax

```
while <condition>:   # condition must evaluate to a boolean
    <code block>     # the indent defines the loop's code block
```

6.3.2 example

```
>>> import random
>>> warm = 20
>>> temperature = random.randrange(5, 30)
>>> while temperature <= warm:
...     print('cold')
...     temperature = random.randrange(5, 30)
cold
cold
cold
```

6.3.3 visualising execution

Tip: Use while if you don't when you only know a loop will terminate in a given condition.

6.3.4 loop keywords

break is a keyword that instructs the interpreter to break out of a loop. *continue* instructs the interpreter to skip the rest of the loop code block and continue with the next loop.

6.4 Practicals

6.4.1 Practical: Loan

A loan repayment plan consists of a balance and monthly interest and repayments.

The loan amount in question is £100. Repayments are made at £20. Interest is charged monthly at %10.

Write a program that prints to screen the remaining balance after every month.

6.4.2 Practical: Shoe Conversion

A UK company wants to export shoes to continental Europe.

It hires you to write a program that prompts the user for a UK size and return the equivalent size it would be in Europe.

Here is a conversion table:

Europe	UK
38	5
39	6
40	7
42	8

6.4.3 Practical: BMI Calculator

The NHS has hired you to create a BMI Calculator.

Write a command line program that asks a user for:

- Weigth in Kilograms
- Height in Meters

Return the bmi result, followed by the users' BMI classification.

BMI Classification

Tip: You will have to do some research online for how to calculate a persons bmi.

BMI	Classification
18.5 or less	Underweight
18.5 to 24.99	Normal Weight
25 to 29.99	Overweight
30 to 34.99	Obesity (Class 1)
35 to 39.99	Obesity (Class 2)
40 or greater	Morbid Obesity

6.4.4 Practical: Turtles Joypad

We want to control the movements of the turtle using instructions from the keyboard. Much like the way you'd control a character in a game.

Place this in a file called *turtle_joypad.py*:

```
import turtle

tess = turtle.Turtle()

while True:
    move = input('\nType a w d s for left up right down (q to exit): ')
    if move == 'a':
        tess.setheading(180)  # west
        tess.forward(10)

    # [ ... put your code here ... ]
```

```
    if move == 'q':
        break
```

CHAPTER 7

Functions

You can think of functions as actions, verbs, or commands and you can think of parameters as adverbs: 'run, quickly'

Functions are special objects that contain code.

When you call them, using special syntax *()*, you execute the code they contain.

```
>>> turtle.forward              # functions have names
<function turtle.forward>
>>> turtle.forward(10)          # actioned by use of '()'s
```

All turtle instructions are examples of calling functions attached to the turtle object.

print is another function:

```
>>> print('hello')
```

print simply prints its parameter to the console.

Tip: Functions and methods are very similar. Methods exist on objects however functions stand alone.

7.1 Function objects

A function like everything in Python is an object. Function objects are different in that they contain blocks of code.

Functions help in letting programmers organise and reuse code. They help create new abstractions.

Function objects have names. The name is assigned at the same time you define a function.

7.1.1 defining

Creating function objects requires special syntax:

```
>>> def my_function():
...     print('hello')     # Note 4 space indentation
...
>>> type(my_function)
function
```

The `def` keyword is followed by the function object name, followed by () and then a colon.

example:

```
def going_nowhere():
    turtle.forward(50)
    turlle.backward(50)
```

Note:

- The body of a function is the following block of code.
- A block is defined by a colon, and one or more indented lines.
- The indents are 4 spaces. The block ends on the first non indented line. (Take care to use spaces and not tabs for indenting)

7.1.2 Usage

We 'call' functions by adding *()* at the end of their names. This is syntax unique to functions. It means action the function objects' code block.

7.1.3 IndentationError

Indentation is the number of spaces from the left hand side. In python it defines blocks of code.

If you get this kind of error:

```
>>> def awef():
... print('hi')
  File "<stdin>", line 2
    print('hi')
        ^
IndentationError: expected an indented block
```

It simply means the indentation wrong. Here the programmer has forgotten to add 4 spaces on the new line after the colon.

7.2 Arguments

We saw names generalise code and eases code reuse. This is also true of functions that take arguments.

Compare this function without arguments:

```
def draw_right_angle():
    turtle.forward(10)
    turtle.left(90)
    turtle.forward(10)
```

to this one with arguments:

```
def draw_right_angle(length):
    turtle.forward(length)
    turtle.left(90)
    turtle.forward(length)
```

The second function is more flexible. It can be used to move by any length.

The argument acts as a *variable* only defined inside the function's code block.

Functions can have many arguments:

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

7.3 Function Scope

We have seen two ways to add to a given namespace:

1. An assignment statement adds a name that references an object.
2. A function definition associates a name with an object of type function.

Functions however create a namespace for the code it contains.

We will use pythontutor to exercise visualising program execution.

Step through each line of code in the browser.

Notice that when execution enters a function, a new 'frame' is created.

The interpreter creates a new namespace associated with this frame. It is isolated from the 'parent' frame's namespace. This namespace is emtpy unless parameters are passed.

Tip: A namespace and a frame are different objects. For the purpose of this course however think of them as the same.

7.4 Exercises

7.4.1 Shapes with Arguments

Reopen `shapes.py` and define every shape as a function with sensible arguments.

Consider whether this make the code more modular, readable, reusable and general?

7.4.2 House

Refactor (rewrite) your house code as a function that uses two other functions.

7.4.3 Conversion Programs

This exercise assumes you have completed the *about_functions* koans.

For each conversion function you completed in the Koans, write a simple command line program that prompts the user for input and returns the result.

For example with the function convert_to_miles, create a file named *convert_miles_to_kilometers.py* and put your code in there.

Expect users to be able to run this kind of dialog:

```
> python convert_miles_to_kilometers.py         # user runs program
Please enter miles to convert: 34               # user enters 34
34 miles corresponds to about 54.4 kilometers
>
```

Do the same for celsius to farenheit.

CHAPTER 8

Data Structures

Data structures contain other objects. We will look at two: *lists*, *dictionaries*.

Typical methods defined on data structures are ones that allow access and update items within it.

As always first we explore how to create objects using literals and constructors, we then examine some methods typical of each object.

Second, we often want to do something for each item in a data structure. This involves 'iterating' over it. We do this using the for loop.

8.1 Lists

A list object contains ordered items.

8.1.1 Creation

```
>>> ['John', 'Eric', 'Michael', 'Terry']          # literal
>>> list('abc')                                   # `list` transform a string
['a', 'b', 'c']
>>> type([1, 2, 3])
<class 'list'>
```

8.1.2 Extraction & Update

```
>>> abc = ['a', 'b', 'c']
>>> abc[0]                          # extract item
'a'
>>> abc[2] = 'd'                    # update item
>>> abc
['a', 'b', 'd']
```

Tip: Lists are 0 indexed. The first item is at index 0, the second at 1...

If you ask for an item that is outside of the list's length you will get an *IndexError*.

8.1.3 range

The *range* function combined with the *list* constructor is a fast way to create a list with a specified number of increasing integers.

```
>>> list(range(3))    # think: give me numbers up to 3
[0, 1, 2]
```

It provides a convenient shortcut to do things a certain number of times.

```
>>> for i in range(2):
...     print('hi')
'hi'
'hi'
```

8.2 Dictionaries

Dictionaries contain key value mappings.

They can be used to collect information (data) about something. Here we use a dictinary to represent a Person.

8.2.1 creation

```
>>> {'name': 'Brian', 'age': 23, 'sex': 'M'}              # literal
>>> dict([('name', 'Brian'), ('age', 23), ('sex', 'M')])  # constructor
```

8.2.2 extraction & update

Special syntax: *<dict-name>[<key>]* for extracting and updating an attribute.

```
>>> person = {'name': 'Brian', 'age': 23, 'sex': 'M'}
>>> person['name']                          # extract value
'Brian'
>>> person['name'] = 'Naomi'                # update value
>>> person['name']
'Name'
```

If you request a non-existent key you get a *KeyError*.

8.3 Nesting

Data structures can include any type of object including other data structures.

Here is a list of dictionaries:

```
>>> persons = [
        {'name': 'Naomi', 'age': 32, 'sex': 'F', 'status': 'Single'},
        {'name': 'Jane', 'age': 29, 'sex': 'F', 'status': 'Married'},
        {'name': 'Brian', 'age': 23, 'sex': 'M', 'status': 'Single'}
    ]
```

Nested data structures are extremely common.

Think how this could be useful for example to store information about all students in a class.

8.4 The *for* loop

Use *for* to iterate over each item in a given list.

Here by iterate through a list of *str* objects we change the colour of our turtle alex.

```
from turtle import Turtle, exitonclick

alex = Turtle()

for a_colour in ["yellow", "red", "purple", "blue"]:
   alex.color(a_colour)
   alex.forward(50)
   alex.left(90)

exitonclick()
```

8.4.1 Refactoring *square*

We refactor *square* combining *range* with a for loop.

```
def square(side):
    for i in range(4):
        turtle.forward(side)
        turtle.left(90)
```

Drawing a square is reduced to repeating the same action four times.

Thanks to the for loop our definition of a square in code:

- is shorter and more readable.
- communicates an insight into the geometry of a square.

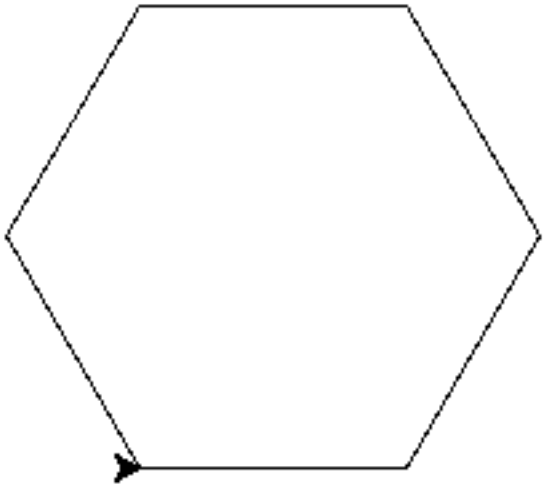
8.5 Exercises

8.5.1 Refactor *shapes.py*

Refactor all the shapes in *shapes.py* and make good use of loops where you can.

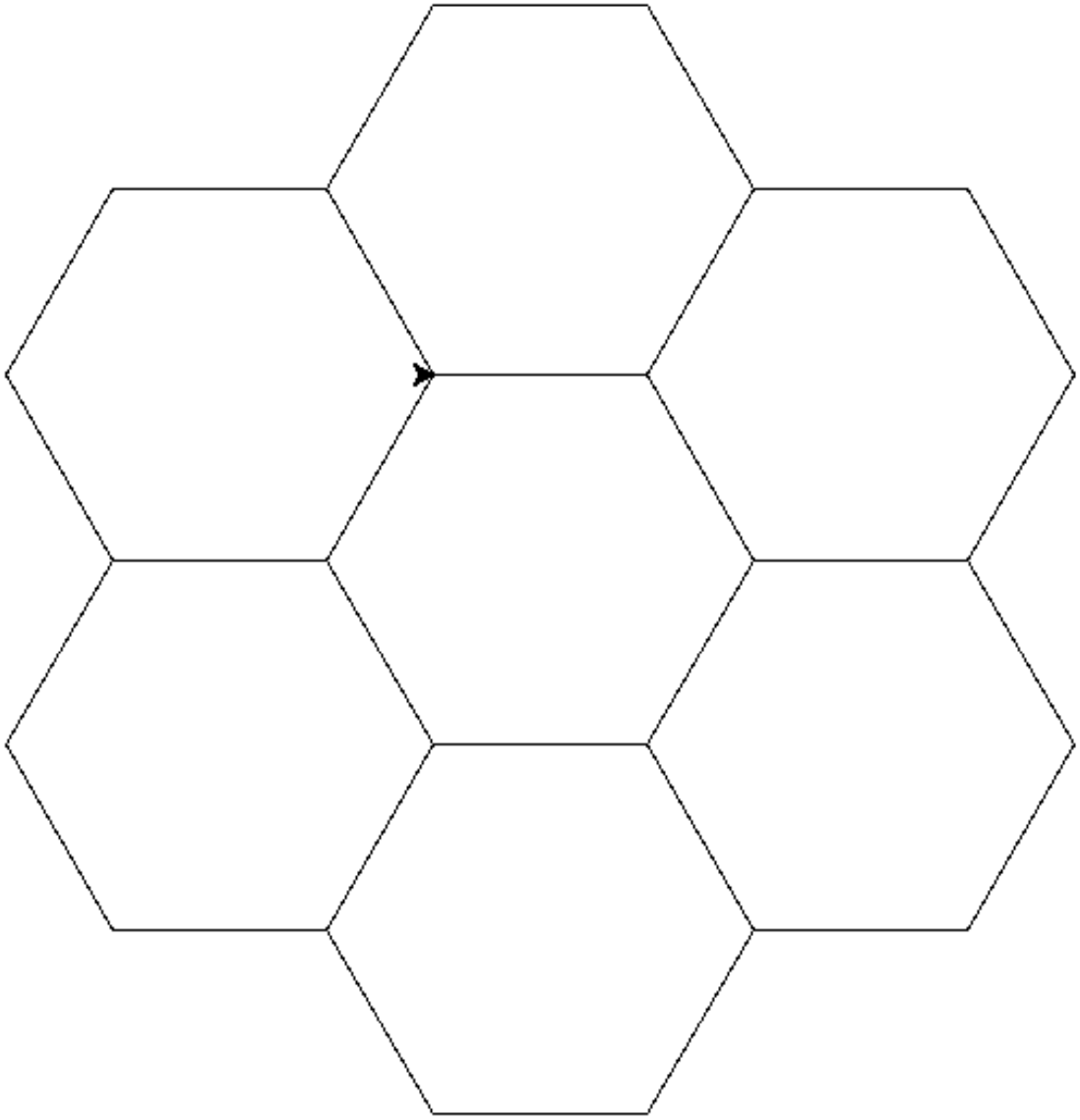
8.5.2 Hexagon

Write code that draws this:



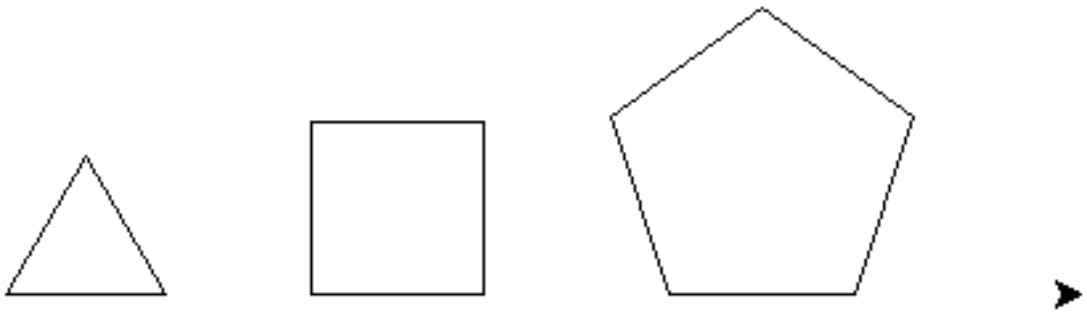
8.5.3 Honeycomb

Write code that draws this:



8.5.4 Any Shape

Write code that can draw any shape like this:



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

8.5.5 Practical: Paper Sissors Rock

Steps:

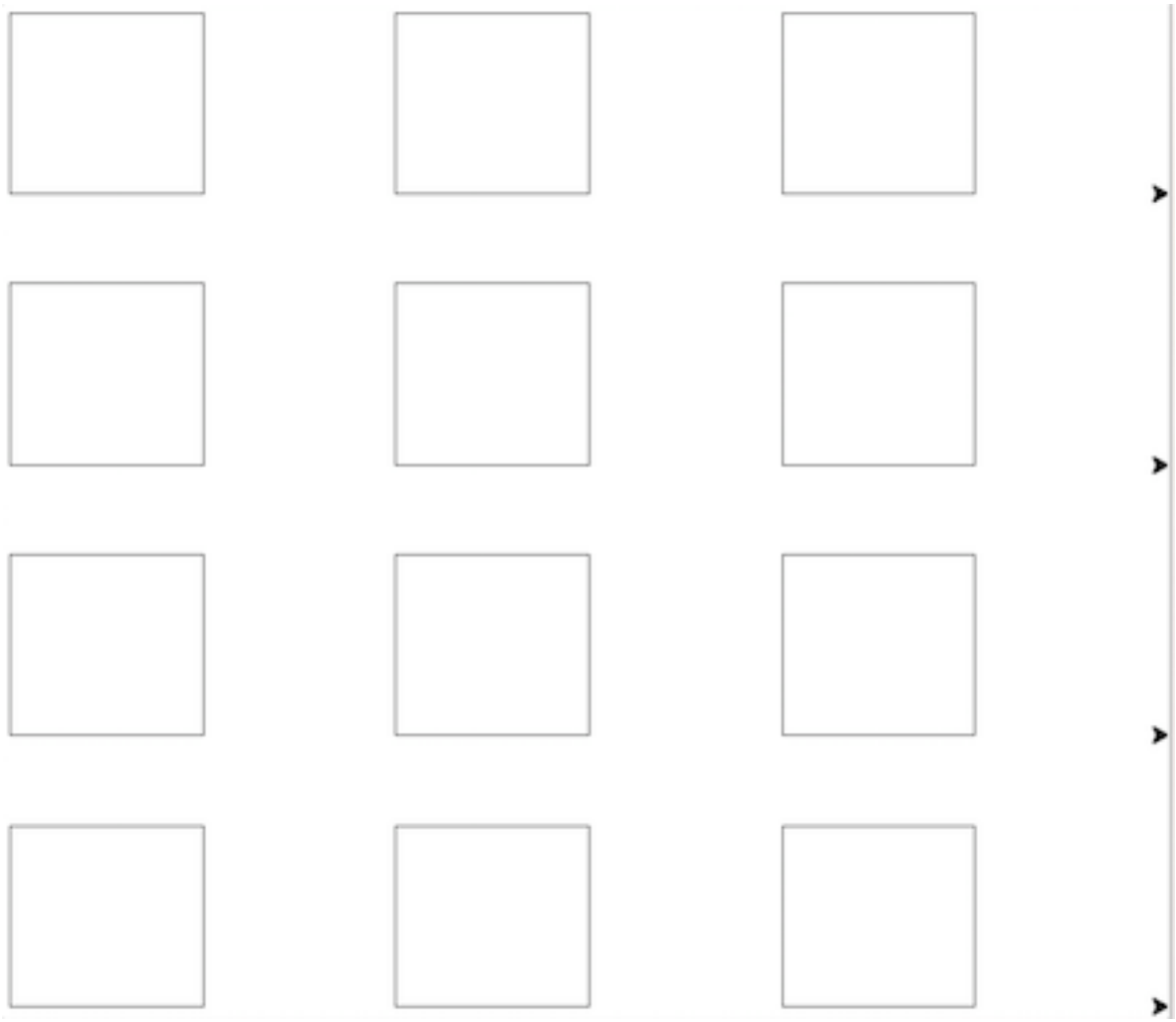
1. user inputs either paper, sissors or rock.
2. computer randomly chooses one too.
3. print outcome according to the rules of the game:
   - If user chose 'paper' and computer chose 'rock', then print 'rock wins'
   - if user chose 'sissors' and computer chose 'paper' then print 'sissors wins'
   - ... and so on ...
4. Exit

You will need to use some randomness:

```
>>> import random
>>> random.choice(['a', 'b', 'c'])
```

8.5.6 Looping *turtles*

Using the following as template draw this:



Put the following in a file called *turtle_queue.py* and finish off the program.

```
import turtle

number_of_turtles = 4

turtles = []
for _ in range(number_of_turtles):
    turtles.append(turtle.Turtle())

# position point of origin at bottom left of window
turtle.setworldcoordinates(0, 0, 600, 600)

for i, turtle_ in enumerate(turtles):
    turtle_.up()

# Evenly space out the turtles
for i, turtle_ in enumerate(turtles):
    ypos = 600 / number_of_turtles * i
    turtle_.setpos(0, ypos)

for i, turtle_ in enumerate(turtles):
    turtle_.down()
```

```
###################################
# Your turn! Enter your code here #
###################################
```

CHAPTER 9

Conclusions

9.1 Programming

The constructs we have learnt (loops, conditions, data structures) mean that we are far more expressive as programmers.

Combined with abstractions we can compose and recompose new programs.

Building on our previously defined concept of a house we now use repetition to define a row of houses.

```
def row_of_houses(number, size):
    for i in range(number):
        house(size)
        turtle.forward(size)
```

This is how complex and useful programs are built.

9.2 Abstractions

We have gone from understanding this:

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

to programming like this:

```
def square(side):
    for i in range(4):
        turtle.forward(side)
        turtle.left(90)
```

Why is the second version better than the first?

Computers are complex. Even the smallest operation hides layers of incredible complexity. Programming is not only about getting a computer to do things. It is about writing code that is useful to humans.

Good programming is harnessing complexity by writing code that rhymes with our intuitions. Good code is code that we can use with a minimal amout of context and already be productive.

By calling:

```
>>> square(100)
```

The above code called *square* can be understood even by a non programmer. Intuition helps because the code is defined at the appropriate level of abstraction over the complex details for understanding to take place.

The two major advantages are:

- detail and complexity is hidden.
- the definition of the function object called *square* is shorter clearer and truer to its mathematical (conceptual) definition.

This course illustrates that creative programming is about constructing useful abstractions. It is also about exercising your intuition to make you more productive.

9.3 Design

We have gone from step by step instructions to defining blocks of code in such a way as to define higher level concepts.

Defining reusable components and the ability to repeat them is immensely powerful.

Think of everything you can make from Lego bricks. Minecraft is a world build with cubes. In the real world think of all the components and repetition you typically find in a skyscraper.

This is where programming starts to become creative. You can define the universe of things that is of interest to you.

9.4 Exercises

9.4.1 A Text editor

Think about the objects that you'd have to use to reprsent editing text.

9.4.2 Your Project

Programmers model other domains. Think of an area where you are expert and how you might code it.

What objects, functions and variables would need to be defined?

CHAPTER 10

Koans (Optional)

Koans are puzzles or exercises that are a great way to reinforce your learnings of a programming language's constructs.

Find instructions on how to download and run them below.

10.1 Making assertions

The koans use the keyword *assert* a lot. When you assert something you are stating that it must be true.

In python true and false are represented by the keywords True and False.

assert passes silently when it is followed by True but throws an Error when followed by False:

```
>>> assert True
>>> assert False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>>
```

In the Koans you have to find answers that evaluate to True for the *assert* to pass.

Visit the appendix on windows for getting started.

Using your intuition try to complete the about_asserts koans.

Making Assertions:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_asserts
```

10.2 The Koans

Tip: When confused by code, break it up and use the interactive interpreter to experiment. Formulate assumptions and test them.

Integers:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_integers
```

Strings:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_strings
C:\Users\greg-lo> python3 contemplate_koans.py about_string_manipulation
```

Conditionals:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_if_statements
```

Functions:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_functions
```

Data Structures:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_lists
C:\Users\greg-lo> python3 contemplate_koans.py about_dictionaries
```

Loops:

```
C:\Users\greg-lo> python3 contemplate_koans.py about_loops
```

10.3 Instructions

10.3.1 Setup

We need to:

1. Download the koans zip file.
2. Unzip it
3. Move the unzipped folder from Downloads to your home directory (for me its *C:Usersgreg-lo*)

Now open *cmd.exe*:

```
C:\Users\greg-lo>
```

Change directory (*cd*) into the python-koans-master directory:

```
C:\Users\greg-lo> cd python-koans-master
C:\Users\greg-lo\python-koans-master>
```

Note the updated location in the prompt.

You are setup!

10.3.2 Running

You run the koans by calling the *python3* interpreter on the *contemplate_koans.py* file followed by a name such as *about_asserts*:

Now we are ready to execute the contemplate_koans.py program:

```
C:\Users\greg-lo> python contemplate_koans.py about_asserts
```

The above instruction is understood as calling the python program and passing in two parameters: a file name 'contenplate_koans.py' and some text 'about_asserts'.

The output should be similar to this:

```
Thinking AboutAsserts
  test_assert_truth has damaged your karma.

You have not yet reached enlightenment ...
  AssertionError: 0 is not true

Please meditate on the following code:
  File "/Users/greg/TEACHING/python_koans/koans/about_asserts.py", line 13, in test_assert_truth
    assert False  # replace with True


You have completed 0 koans and 0 lessons.
You are now 77 koans and 9 lessons away from reaching enlightenment.
```

Note the section that asks you to mediate on a file with a line number.

10.3.3 Answering

Open this file in SublimeText. Find SublimeText in the Start search prompt.

Open the file as specified by the output of *contemplate_koans*. In the above case:

- open *C:Usersgreg-lopython-koans-masterkoansabout_asserts.py*
- Go to line 13 and replace *False* with *True*.
- Save the file.
- Rerun the Koans

You should find that one line has gone Green and you now have a new challenge.

Tip: Arrange the windows on your screen so that you have your text editor on one side and two ‘cmd.exe‘s on the right one above the other.

Have the command prompt open in one for running the koans.

Have the python interpreter in the other for experimenting with code.

CHAPTER 11

Appendix A: Windows

You interact with python using the *cmd.exe* program.

You can find it by searching in the start prompt.

A shortcut:

1. Press *Windows* + *R* (the two keys together)
2. A search prompt pops up.
3. Type *cmd.exe* and press enter.

11.1 Command line 101

Typically you interact with your operating system using a mouse with certain actions: point, click, drag. Using these you can launch programs and move files.

The *cmd.exe* program offers the same interaction but using typed commands:

When *cmd.exe* launches you get a prompt:

```
C:\Users\greg-lo>
```

The prompt gives you your currnent location followed by a >. Here I am in the directory *greg-lo* which itself is in the directory *Users*. *C* refers the hardrive I am on.

Now you enter the *dir* command:

```
C:\Users\greg-lo> dir
```

This will list all the files and folders in your current directory.

Here are all the commands you need for this course:

- cd - change directory
- dir - list the directory's contents
- copy - copy a file or a directory
- move - move a file or a directory
- mkdir - make a directory
- del - delete a file or directory
- unzip - unzip a zipped (compressed) file

Tip: There is nothing here that you aren't familiar doing with the mouse. If necessary use your mouse to orientate yourself.

11.2 The Python Interpreter

When you install python in windows it gives you the option to add the executable (*python.exe*) to your system path.

Unfortunately we need to specify the full path each time: *Python34python.exe*.

```
C:\Users\greg-lo>python
Python 3.4.2 (v3.4.2:8711a0951384, Sep 21 2014, 21:16:45) [MSC v.1600 32 b
it (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Tip: At first it is normal to confuse the command line and the python interpreter. Python instructions don't run in the command shell and shell commands don't work in the interpreter.

The interpreter has >>> as its prompt

The command shell has the file path eg *C:Usersgreg-lo>*

CHAPTER 12

Appendix B: Debugging

Exceptions occur when the interpreter can't carry out a given instruction. The type of error (Exceptions are objects) communicates what is wrong.

We stress that most of programming is error driven. Don't think of errors negatively rather they are problem solving opportunities.

Debugging is working out what went wrong and fixing it.

Learn to be guided by Errors, and use debugging tools to master programming.

Here we explore some common errors and then we introduce *pdb* the python debugger.

12.1 Errors

Errors always tell you what when wrong but not always why.

Read errors, first using intuition then by debugging and research.

Tip: You need to learn how to find information. Always read Errors and use your intuition, then Google. If that hsn't helped only then ask an expert.

With time many errors map to solutions instantly.

12.1.1 AttributeError

An *AttributeError* means the interpreter can't find the name you have asked for on the object.

```
>>> import turtle
>>> turtle.shp('waef')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'shp'
```

Here the programmer has misspelt shape.

12.1.2 SyntaxError

Learning a language involves making many syntax (grammatical) errors.

A function defined badly:

```
>>> def print_hi:
  File "<stdin>", line 1
    def print_hi
                ^
SyntaxError: invalid syntax
```

Parentheses *()* are required after the name and the ending colon *:*.

```
>>> def print_hi():
    print('hi')
```

No error, *print_hi* is properly defined.

12.2 pdb

pdb is the python debugger. You can freeze execution at a particular point in time, step through it, examining objects as you go.

To execute code with *pdb*:

```
python3 -m pdb my.py
```

You can also pause the execution at any time by placing this line into your code:

```
import pdb; pdb.set_trace()
```

When you run your code normally (*python my.py*) the interpreter will break at the that line of code.

Type *h* to get a list of all the commands. The important ones for now are:

Move along the execution timeline:

- *l* print lines of code surrounding cursor
- *n* execute next line
- *s* step into a line. Typically used for entering functions.
- *c* continue till the end of the program (or next break point).

Inspect the current location:

- *w* print frames on the stack at current position
- *u* go up a frame in stack
- *d* go down a frame in the stack

To exit: * *q* exit the debugger. Will terminate program execution.

Tip: On any error or exeception enter a *import pdb; pdb.set_trace()* on the line preceeding your program terminating. Run the program, then inspect what went wrong.

12.2.1 example

We will use pythontutor hand in hand with pdb to exercise visualising program execution.

Put this code into a file named *my.py*:

```
x = 1
y = 2
success = 'works'
failure = 'broken'

def inc(p):
    incremented = p + 1
    return incremented

def print_result(result):
    if result:
        print(success)
    else:
        print(failure)

inc_x = inc(x)
print_result(inc_x == y)
```

Execute with:

```
python3 -m pdb my.py
```

pdb starts program and pauses at first line:

```
> /Users/greg/my.py(1)<module>()
-> x = 5
(Pdb)
```

Executing *l* results in:

```
(Pdb) l
  1  ->     x = 5
  2         y = 6
  3
  4         def f():
  5             z = 4
  6             total = sum(x, y, z)
  7             return total
  8
  9         print('hi')
 10         print(f())
[EOF]
```

Step through each line of code keeping.

Ensure you explore the two frames when you enter the f functions' frame.

Tip: We have used a python code visualiser in a similar way.

12.3 koans & *pdb*

pdb is a great tool to understand code. Here we will apply it to our koans.

Enter:

```
import pdb; pdb.set_trace()
```

In a koan method that caused you difficulty.

Step through the execution of the code. When you are done type *c* to continue normal execution.

CHAPTER 13

Appendix C: Classes

We are now going to bring what we have learnt about object oriented programming together as we define our own object type using classes.

Classes are how programmers define objects that make new objects. Think of them as object templates or factories.

Tip: The Koans are structured as classes with each koan as a method.

13.1 Defining & usage

Much like we defined functions lets define a class.

A *python.py* file contains:

```
class Python():
    """ A class that represents a snake """

    def __init__(self, name, sex, age, length):
        self.name = name
        self.sex = sex
        self.age = age
        self.length = length

    def move(self):
        print("{} moves".format(self.name))

    def eat(self):
        """ a snake gets longer when it eats """
        self.lenth = self.legnth + 1

    def starve(self):
        """ a snake shorter when it starves """
        # is there a bug here?
        self.lenth = self.legnth - 1
```

13.1.1 class object

Lets introspect the new type of object:

```
>>> from python import Python
>>> type(Python)
<class 'type'>
>>> dir(Python)
[ ... many methods ... ]
```

13.1.2 instances

A class is like an object instance factory. Here our class makes snakes.

Implicitly it runs the __init__ function as defined on the class.

Creating:

```
>>> john = Python('John', 'M', 15, 4)
>>> jane = Python('Jane', 'F', 4, 6)
```

Introspecting:

```
>>> type(john)
<class 'python.Python'>
>>> dir(john)
[ ... many methods ... ]
```

Note we get *move*, *starve*, and *eat* which we defined, but we also get many methods others.

Tip: The other methods are those found when executing *dir(object)*

13.1.3 methods

```
>>> Python.move
<function Python.move at 0x10f9b6840>
>>> john.move
<bound method Python.move of <python.Python object at 0x10fb04898>>
```

A function and a method are very similar. A function can stand alone, a method however is 'bound' to an object. When defined methods always take self as their first argument. It is thereby implicit when called.

13.2 snakes

The *__str__* special method is called on an object when we pass it to the *print* function.

We decide that the semantics of printing a python is to show a visual representation of a snake using characters.

Added to definition in *python.py*:

```
class Python():

    def __init__(self, name, sex, age, length):
        self.name = name
[...]

    def __str__(self):
        body = '=' * self.length
        return "{}>".format(body)
```

results:

```
>>> from python import Python
>>> john = Python('John', 5)
>>> print(john)
~-=====%>
```

13.3 special methods

13.4 Exercises

13.4.1 attack

Decide on the semantics of a python attacking another object.

Implement your decision by defining a new method.

13.4.2 __add__

Lets define another special method to exploit the nice syntax python gives us.

Decide on the semantics of 'adding' pythons together.

Implemnent by defining your __add__ method on the Python class.

CHAPTER 14

Appendix D: Resources

There are many resources freely available on the web for further learning.

One of the strengths of Python relative to other languages is the diversity of of applications it has.

Do find one area of interest and make it your own.

14.1 Documentation

https://docs.python.org/3/

The important ones are:

- Tutorial
- Library Reference - Practical
- Language Reference - Academic, authoritative.

14.2 Tutorials

- start with these:
  - http://interactivepython.org/runestone/static/thinkcspy/toc.html
  - http://opentechschool.github.io/python-beginners/en/index.html
  - http://learnpythonthehardway.org/ Not hard, hands on
- Popular:
  - http://www.codecademy.com/en/tracks/python
  - http://inventwithpython.com/
  - http://www.diveintopython3.net/

14.3 Online Courses

Python today is widely taught in University. Here are some execelent resources.

- Udacity - CS101

- MIT Intro to Computer Science and Programming (MITOpenCourseware)
- Introduction to Interative Programming in Python (Coursera)
- Learn to Program: The Fundamentals (University of Toronto, Coursera)

14.4 Applied

The best way to consolidate learning programming is to find a domain that is of interest and learn more about programming it.

Here are some resources that will inspire.

14.4.1 Natural Language Processing

Natural Language processing refers to tools used to parse and add semantics to human language. Talking with computers.

- http://www.nltk.org/
- http://www.nltk.org/book/
- Python 3 Text Processing with NLTK 3 Cookbook

14.4.2 Web development

- https://docs.djangoproject.com/en/ Django Tutorial
- http://www.obeythetestinggoat.com/ To become a pro!

14.4.3 Games

- http://www.pygame.org/wiki/tutorials

14.4.4 3D Graphics

- Blender
  - http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts
  - http://www.blender.org/api/blender_python_api_2_72_1/
- Maya
  - http://zurbrigg.com/maya-python

14.4.5 Electronics

Raspberry Pi is currently the center of focus for fun with home electronics.

- Make: Sensors: A Hands-On Primer for Monitoring the Real World with Arduino and Raspberry Pi
- Make: More Electronics: Journey Deep Into the World of Logic Chips, Amplifiers, Sensors, and Randomicity
- Raspberry Pi Home Automation with Arduino

- Raspberry Pi for Secret Agents
- Raspberry Pi Cookbook for Python Programmers
- OReilly RPi
- Getting Started with Sensors: Measure the World with Electronics, Arduino, and Raspberry Pi

14.4.6 Geospatial (GIS)

- http://geospatialpython.com/
- http://qgis.org/en/site/
- http://docs.qgis.org/2.0/en/docs/pyqgis_developer_cookbook/

14.4.7 Biology

- http://biopython.org/DIST/docs/tutorial/Tutorial.html

14.4.8 Maths

- http://www.sagemath.org/doc/thematic_tutorials/
- http://www.sagemath.org/doc/tutorial/

14.4.9 Forensics

- Violent Python (book)
- Grey Hat Python (book)

14.4.10 Mobile

- http://kivy.org/docs/tutorials/pong.html

14.4.11 Data Science

- Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (book)
- http://pandas.pydata.org/pandas-docs/dev/tutorials.html

14.4.12 Machine Learning

- Scikit-Learn: Machine Learning in Python
- http://scikit-learn.org/stable/tutorial/index.html
- Practical Data Science Cookbook (book)
- Building Probabilistic Models with Python (Book)

14.5 Course References

Some materials that inspred this course.

- Open Tech Schoool
  - http://opentechschool.github.io/python-beginners/en/index.html
- How to Design Programs
  - http://htdp.org/
- Structure and Interpretation of Computer Programs
  - https://mitpress.mit.edu/sicp/full-text/book/book.html