Python: A whirlwind tour

In this tutorial, we'll introduce the basic concepts of programming, which will be central to the programs that you will run on your robot. There are many different languages in which computers can be programmed, all with their advantages and disadvantages, but for the Student Robotics competition we use one called Python 2.6. We chose it because it's good for beginners, but also elegant and powerful.

Sprinkled through the tutorial are exercises. The first ones for each section should be quite easy, while the higher-numbered exercises will be harder. Some will be very hard; try these if you're up for a challenge.

Before we begin: a word on learning. The way that you learn to code is by doing it; make sure you try out the examples, fiddle with them, break them, try at least one exercise from each section. So, on with the tutorial!

Using an interpreter

To run Python programs you need a something called an interpreter. This is a computer program which interprets human-readable Python code into something that the computer can execute. There are a number of online interpreters that should work even on a locked-down computer such as you will probably find in your college.

If your computer has a compatible browser, go to `http://repl.it` and click Python. Enter your program in the box on the left, and click the "run" button.

If your browser isn't compatible, another good online interpreter can be found at `http://codeskulptor.org`. It's very similar; simply enter your program into the left pane and click the play button to run it. The output will appear in the right pane.

Whichever you choose, test it with this classic one line program:

```
print "Hello World!"
```

The text `Hello World!` should appear in the output box.

There's nothing particularly wrong with online interpreters for our needs, but if you want to use Python for something more advanced you'll want an interpreter

which runs directly on your computer. Mac OS X and Linux come with one by default (just type `python` at the terminal), and you can download the Windows interpreter from http://python.org/download (try Portable Python (http://portablepython.com) if you can't install programs on your computer).

Statements

A statement is a line of code that does something. A program is a list of statements. For example:

```
x = 5
y = (x * 2) + 4
print "Number of bees:", y - 2
```

The statements are executed one by one, in order. This example would give the output `Number of bees: 12`.

As you may have guessed, the `print` statement displays text on the screen, while the other two lines are simple algebra.

Strings

When you want the interpreter to treat something as a text value (for example, after the `print` statement above), you have to surround it in quotes. You can use either single (') or double (") quotes, but try to be consistent. Pieces of text that are treated like this are called 'strings'.

Comments

Placing a hash (`#`) in your program ignores anything after the hash.

For example:

```
# This is a comment
print "This isn't."  # But this is!
```

You should use comments whenever you think that it is not completely clear what a statement or block of statements does, especially as you are working in teams! Also bear in mind the varying coding skills of your team. You might be the best coder in your team, but what if you were taken ill the day before the competition, and your team-mates had to fix your code?

Comments are also useful for temporarily removing statements from your code, for testing:


```
x = 42
#x = x - 4
print "The answer is", x
```

This example would output `The answer is 42`, as the subtraction is not executed.

Variables

Variables store values for later use, as in the first example. They can store many different things, but the most relevant here are numbers, strings (blocks of text), booleans (`True` or `False`) and lists (which we'll come to later).

To set a variable, simply give its name (see Identifiers, below), followed by `=` and a value. For example:

```
x = 8
my_string = "Tall ship"
```

You can ask the user to put some text into a variable with the `raw_input` function (we'll cover functions in more detail later):

```
name = raw_input("What is your name?")
```

Concept: Identifiers

Certain things in your program, for example variables and functions, will need names. These names are called 'identifiers', and must follow these rules:

- Identifiers can contain letters, digits, and underscores. They may not contain spaces or other symbols.
- An identifier cannot begin with a digit.
- Identifiers are case sensitive. This means that `bees`, `Bees` and `BEES` are three different identifiers.

Exercises: Variables and Mathematics

Average calculator

The first two lines of this program put two numbers entered by the user into variables `a` and `b`. (The `input` function is like `raw_input`, but returns a number (e.g. `42`) when you enter one, rather than a string (like `"42"`).) Replace the

comment with code that averages the numbers and puts them in a variable called `average`.

```
a = input("Enter first number: ")
b = input("Enter second number: ")

# Store the average of a and b in the variable 'average'

print "The average of",a,"and",b,"is",average
```

Run your code and check that it works.

Distance calculator

Write a program which uses `input` to take an X and a Y coordinate, and calculate the distance from (0, 0) to (X, Y) using Pythagoras' Theorem. Put the code into an interpreter and run it. Does it do what you expected?

Hint: you can find the square root of a number by raising it to the power of 0.5, for example, `my_number ** 0.5`.

Extension: can you adapt the program to calculate the distance between any two points?

Booleans and `if` statements

A boolean value is either `True` or `False`. For example:

```
print 42 > 5
print 4 == 2
```

Output:

```
True
False
```

`<` and `==` are operators, just like `+` or `*`, which return booleans. Others include `<=` (less than or equal to), `>`, `>=` and `!=` (not equal to) (see the Operators appendix). You can also use `and`, `or`, and `not`.

`if` statements execute code only if their condition is true. The code to include in the `if` is denoted by a number of indented lines (see the concept section on code blocks). To indent a line, press the tab key or insert four spaces at the start. You can also include an `else` statement, which is executed if the condition is false. For example:


```
name = raw_input("What is your name?")
if name == "Tim":
    print "Hello Tim."
    print "You've got an email."
else:
    print "You're not Tim!"

print "Python rocks!"
```

If you typed "Tim" at the prompt, this example would output:

```
Hello Tim.
You've got an email.
Python rocks!
```

Having another `if` in the `else` block is very common:

```
price = 50000 * 1.3
if price < 60000:
    print "We can afford the tall ship!"
else:
    if price < 70000:
        print "We might be able to afford the tall ship..."
    else:
        print "We can't afford the tall ship. :-("
```

So common that there's a special keyword, `elif`, for the purpose. So, the following piece of code is equivalent to the last:

```
price = 50000 * 1.3
if price < 60000:
    print "We can afford the tall ship!"
elif price < 70000:
    print "We might be able to afford the tall ship..."
else:
    print "We can't afford the tall ship. :-("
```

Both output:

```
We might be able to afford the tall ship...
```


Concept: Code blocks and indentation

In the previous section, you probably noticed that the statements 'inside' the `if` statements were indented relative to the rest of the code. Python is reasonably unique in that it cares about indentation, and uses it to decide which statements are referred to by things like `if` statements.

If you don't indent your code in other programming languages, it will run just fine, and any poor soul who has to read your code afterwards will hunt you down and hit you around the head with a large, wet fish. In Python, you'll just get an error, which we're sure you'll agree is preferable.

A group of consecutive statements that are all indented by the same distance is called a block. `if` statements, as well as functions and loops, all refer to the block that follows them, which must be indented further than that statement. An example is in order. Let's expand the first `if` example:

```
name = raw_input("What is your name?")
email = "Bank of Nigeria: Tax Refund"
if name == "Tim":
    print "Hello Tim."
    if email != "":
        print "You've got an email."

        # (blocks can contain blank lines in the middle)
        if email != "Bank of Nigeria: Tax Refund":
            print "Looks legitimate, too!"
    else:
        print "No mail."

else:
    print "You're not Tim!"

print "Python rocks."
```

Output (for "Tim" as before):

```
Hello Tim.
You've got an email!
Python rocks.
```

To find the limits of an `if` statement, just scan straight down until you encounter another statement on the same indent level. Play around with this example until you understand what's happening.

One final thing: Python doesn't mind *how* you indent lines, just so long as you're consistent. Some text editors insert indent characters when you press tab; others

insert spaces (normally four). They'll often look the same, but cause errors if they're mixed. If you're using an online interpreter, you probably don't need to worry. Otherwise, check your editor's settings to make sure they're consistent. Four spaces per indent level is the convention in Python. We'll now move on from this topic before that last sentence causes a flame war.

Exercises: `if` Statements and Blocks

So many `if`s

Without running it, work out what output the following code will give:

```
some_text = "Duct Tape"
if 5 > 4:
    print "Maths works."
    if some_text == "duct tape":
        print "The case is wrong."
    elif some_text == "Duct Tape":
        print "That's right."
    else:
        print "Completely wrong."
else:
    print "Oh-oh."
```

Run the code and check your prediction.

Age Discrimination Tool

Write a program that asks the user for their age, and prints a different message depending on whether they are under 18, over 65, or in between.

Lists

Lists store more than one value in a single variable, and allow you to set and retrieve values by their position ('index') in the list. For example:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
print shopping_list[0]
shopping_list[3] = "Magazine"
print shopping_list[2]
print shopping_list[3]
```

Output:


```
Bread
PNP Transistors
Magazine
```

Notice that the indices start at 0, not 1. There is a sensible, technical explanation for this that is beyond this tutorial's scope. Also note that because of this, the last element of this four-element list is at index 3. Attempting to retrieve `shopping_list[4]` would cause an error.

You can find out the length of a list with the `len` function, like so:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
print "There are", len(shopping_list), "items on your list."
```

Finally, you can add a value to the end of a list with the `append` method:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
shopping_list.append("Mince pies in October")
print shopping_list
```

The values in a list can be of any type, even other lists. Also, a list can contain values of different types.

There are various other useful data structures that are beyond the scope of this tutorial, such as dictionaries (which allow indices other than numbers). You can find out more about these at http://docs.python.org/tutorial/datastructures.html.

`while` loops

The `while` loop is the most basic type of loop. It repeats the statements in the loop while a condition is true. For example:

```
x = 10
while x > 0:
    print x
    if x == 5:
        print "Half way there!"

    x = x - 1

print "Zero!"
```

Output:

```
10
9
8
7
6
5
Half way there!
4
3
2
1
Zero!
```

The condition is the same as it would be in an `if` statement, and the block of code to put in the loop is denoted in the same way, too.

`for` loops

The most common application of loops is in conjunction with lists. The `for` loop is designed specifically for that purpose. For example:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
for x in shopping_list:
    print "[ ]", x
```

The code is executed once for each item in the list, with `x` set to each item in turn. So, the output of this example is:

```
[ ] Bread
[ ] Milk
[ ] PNP Transistors
[ ] Newspaper
```

Unfortunately, this method doesn't tell you the index of the current item. `x` is only a temporary variable, so modifying it has no effect on the list itself (try it). This is where the `range` function comes in (see the Calling functions section). An example with numbers:

```
prices = [4, 5, 2, 1.50]
# Add VAT
for i in range(len(prices)):
    prices[i] = prices[i] * 1.20

print prices
```


Output:

```
[4.8, 6.0, 2.4, 1.7999999999999998]
```

Exercises: Lists and Loops

A better average calculator

Write a program which calculates the average of a list of numbers. You can specify the list in the code.

Extension: You can tell when a user has not entered anything at a `raw_input` prompt when it returns the empty string, `""`. Otherwise, it returns a string (like "42.5"), which you can turn into a number with the `float` function. For example:

```
var = raw_input("Enter a number: ")
if var == "":
    print "You didn't enter anything!"
else:
    print "You entered",float(var)
```

Now, extend your program to let the user enter the list of values. Stop asking for new list entries when they do not enter anything at the `raw_input` prompt.

Fizz buzz

Write a program which prints a list of numbers from 0 to 100, but replace numbers divisible by 3 with "Fizz", numbers divisible by 5 with "Buzz", and numbers divisible by both with "Fizz Buzz".

Hint: you might find the `range` function from the next section useful.

Extension: create a list of numbers, and replace a number with "Fuzz" if it is a multiple of any number in the list.

Trees and Triangles

You can concatenate strings in Python with the `+` operator:

```
message = "Hello "
message = message + "World!"
print message
```


Write a program that asks the user for a number, and then prints a triangle of that height, with its right angle at the bottom left. For example, given the number 3, the program should output:

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

Try the same, but with the right angle in the top-right, like so (again, for input 3):

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

Extension: print out a tree shape of the given size. For example, a tree of size 4 would look like this:

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

Calling functions

Functions are pre-written bits of code that can be run ('called') at any point. The simplest functions take no parameters and return nothing. For example, the `exit` function ends your program prematurely:

```
x = 10
while x > 0:
    print x
    x = x - 1
    if x == 5:
        exit()  # not supported in repl.it
```

This will output the numbers 10 to 6, and then stop. Not very useful. However, most functions take input values ('parameters') and output something useful (a 'return value'). For example, the `len` function that we used in the second `for` loop example returns the length of the given list:

```
my_list = [42, "BOOMERANG!!!", [0, 3]]
print len(my_list)
```

Output:

```
3
```

Combined with the `range` function, which returns a list of numbers in a certain range, you get a list of indices for the list (you might want to look back at that second `for` example).

```
my_list = [42, "BOOMERANG!!!", [0, 3]]
print range(len(my_list))
```

Output:

```
[0, 1, 2]
```

The `range` function can also take multiple parameters:

```
print range(5)            # numbers from 0 to 4.
print range(2, 5)         # numbers from 2 to 4.
print range(1, 10, 2)     # odd numbers from 1 to 10
```

Output:

```
[0, 1, 2, 3, 4]
[2, 3, 4]
[1, 3, 5, 7, 9]
```

There are many built-in functions supplied with Python (see appendix). Most are in 'modules', collections of functions which have to be imported. For example, the `math` module contains mathematical functions. To use the `sin` function, we must import it:

```
import math

print math.sin(math.pi / 2)
```


Defining functions

Of course, you'll want to make your own functions. To do this, you precede a block of code with a `def` statement, specifying an identifier for the function, and any parameters you might want. For example:

```
def annoy(num_times):
    for i in range(num_times):
        print "Na na na-na na!"

annoy(3)
```

The output would be three annoying lines of `Na na na-na na!`.

To return a value, use the `return` statement. A rather trivial example:

```
def multiply(x, y):
    return x * y

print multiply(2, 3)
```

Using functions effectively

Without functions, most programs would be very hard to read and maintain. Here's an example (admittedly a little contrived):

```
my_string = "All bees like cheese when they're wearing hats."
x = 0
for c in my_string:
    if c == "a":
        x = x + 1

y = 0
for c in my_string:
    if c == "e":
        y = y + 1
```

Before we explain the example, try and figure out what it does. What do `x` and `y` represent?

Now, let's refine it with functions:

```
def count_letter(str, l):
    x = 0
```

```
    for c in str:
        if c == l:
            x = x + 1

    return x

my_string = "Bees like cheese when they're wearing hats."

x = count_letter(my_string, "a")
y = count_letter(my_string, "e")
```

This version has a number of advantages:

- It's far more obvious what the program does.
- The program is shorter, and cleaner.
- The code for counting letters in a string is in only one place, and can be reused.

The last point has another advantage. There's a bug in this program: uppercase letters aren't counted. It's easy to fix, but in the function version we only have to apply the fix in one place. True, it would only be two places in the original, but in a major program, it could be thousands.

You should try and use functions wherever you see multiple lines of code that are repeated, or find yourself writing code to do the same thing (or a similar thing) more than once. In these situations, look at the relevant bits of code and try to think of a way to put it into a function.

Concept: Scope

When you set a variable inside a function, it will only keep its value inside that function. For example:

```
x = 2

def foo():
    x = 3
    print "In foo(), x =", x

foo()
print "Outside foo(), x =", x
```

Output:

```
In foo(), x = 3
Outside foo(), x = 2
```

This can get quite confusing, so it's best to avoid giving variables inside functions ('local' variables) the same identifier as those outside. If you want to get information out of a function, `return` it.

This concept is called 'scope'. We say that variables which are changed inside a function are in a different scope from those outside.

You can have functions within functions, and this can actually be quite useful. In this situation, each nested function will also have its own scope.

Exercises: Functions

Trigonometry

Write a program that takes as input an angle (in radians) and the length of one side (of your choice) of a right-angled triangle. Print out the length of all sides of the triangle.

You'll need the functions contained in the `math` module (http://docs.python.org/library/math.html). Note that Python uses radians for its angles. If you are not comfortable with radians, you can use the `radians` function in the `math` module to convert to radians from degrees.

Extension: you can return multiple values from a function like so:

```
def foo():
    return 1, 2, 3

x, y, z = foo()
```

Wrap your triangle calculation code in a function.

Greeting

Write a function that takes a name as an input, and prints a message greeting that person.

Average function

Wrap the code for your average calculator from the Lists and Loops exercises in a function that takes a list as a parameter and returns its average.

What to do next

As mentioned at the start, there are loads of Python exercises out there on the Web. If you want to learn some more advanced concepts, there are more tutorials out there too, and http://learnpython.org/ looks like a good choice. Start at the Classes and Objects section.

Appendices

Operators

There are three types of operators in Python: arithmetic, comparison, and logical. I'll list the most important.

Arithmetic

The usual mathematical order (BODMAS) applies to these, just like in normal algebra.

`+`, `-`, `*`, `/` Self-explanatory (if you're having trouble with division, read the first half of this article: http://www.ferg.org/projects/python_gotchas.html)

'`%`' Remainder. For example, `5 % 2` is 1, `4 % 2` is 0.

`**` power (e.g. `4 ** 2` is 4 squared)

Comparison

These return a boolean (`True` or `False`) value, and are used in `if` statements and `while` loops. These are always done after arithmetic.

`==`, `!=` equal to, not equal to

`<`, `<=`, `>`, `>=` less than, less than or equal to, greater than, etc.

`in` returns true if the string on the left is contained in the string on the right. For example:

```
if "car" in "Scarzy's hair":
    print "Of course."
```


Logical

These operators are `and`, `or`, and `not`. They are done after both arithmetic and comparisons. They're pretty self-explanatory, with an example:

```
x = 5
y = 8
z = 2

if x == 5 and y == 3:
    print "True"
else:
    print "False"

print x == 5 or not y == 8          # could use y != 8 instead
print x == 2 and y == 3 or z == 2   # needs brackets for clarity!
```

Output:

```
False
True
True
```

When more than one boolean operator is used in an expression, `not` is performed first (as it works on a single operand). After this, `and` is done before `or`, but you should use brackets instead of relying on that fact, for readability. So, the last line of the example should read:

```
print (x == 2 and y == 3) or z == 2
```

Built-in functions

A lot of functions are defined for you by Python. Those listed at http://docs.python.org/library/functions.html are always available, and are the most commonly used, including `len` and `range`.

Others are contained in modules. To use a function from a module, you must `import` that module, like so:

```
import math
print math.sqrt(4)
```

One of the most useful modules for the moment will be `math` (http://docs.python.org/library/math.html).

Common Errors and mistakes

This is just a selection of common error messages. If you encounter one that isn't on this list, quickly check your code and search the Internet for it.

Syntax Error

This error message appears when you have entered a statement that doesn't obey the forms of the language. For example:

```
def foo(s):
    print s

foo "Hello World!"  # should be foo("Hello World!")
```

Error:

```
File "<stdin>", line 4
    foo "Hello World!"
                    ^
SyntaxError: invalid syntax
```

The output shows a problem with the fourth line, where we've forgotten to place brackets around the string parameter. The arrow indicates the place where the interpreter thinks the problem is. As you can see, this could often be more helpful.

Other causes of syntax errors to look out for are:

- Missing colons at the end of `def`s, `if`s, `for`s, etc.
- Using the wrong number of `=` signs (see the Variables section)
- Missing brackets, for example `x = 5 * (3+2`
- Using a comma (`,`) instead of a period (`.`) as a decimal point (as is common in some European countries)

Name Error

```
x = 5
print X     # wrong case
```

Error:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'X' is not defined
```

This error has occurred because the variable was defined as `x`, but referenced as `X` in uppercase. As previously alluded to, Python distinguishes between cases, so these are two different variables.

This error has a traceback. This would list the functions that the error occurred in, if it was inside a function.

Index Error

If you try to access an element of a list that does not exist, you'll get this error. For example:

```
a = ["Molly", "Polly", "Dolly"]
print a[0]
print a[3]
```

Error:

```
Traceback (most recent call in last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

This example illustrates a common cause. As `a` has three elements, you'd expect it to have a third element. However, in Python, the 'first' element is number 0, the 'second' is number 1, and so on. So, the last element in the array is actually number 2, and element number 3 doesn't exist.

Indentation Error

If you forget to indent some code, **or mix tabs and spaces**, you will get an indentation error. For example:

```
if x < 5:
do_some_stuff()
```

Error:

```
  File "<stdin>", line 2
    do_some_stuff()
                 ^
IndentationError: expected an indented block
```