2 Basic Python Programming: Lists, and defining functions

First, congratulations on completing tutorial 1! Frequently the little details of beginning to learn a programming language trip a person up and it's great to be over the first hurdle. My goal is to get you doing programming related to linguistics as quickly as possible. With this in mind, the next few tutorials will focus on the essentials of Python you'll need. This is not a comprehensive description of Python. My choice of what material to present is primarily based on providing you with a solid foundation that will enable you to do interesting linguistic stuff. However, occasionally I will throw in Python tidbits that are just intended to spice things up. For example, in this tutorial I'll introduce you to a function that will pick a random noun from a list of nouns. This will allow us to write a little random sentence generator and make practicing the essentials a little more interesting.

2.1 Names and Values

In tutorial 1 we worked with two types of values: numbers and strings and we saw how we could assign names to these (you can follow along by typing these statements in the PythonWin Interactive Window):

```
>>> a = 5                  assign the name 'a' to the value 5
>>> b = 7                  assign the name 'b' to the value 7
>>> c = a + b              first determine the value of the right side of the
                           equation (a + b → 5 + 7 → 12) and assign it the name 'c'
>>> c                      now find out the value of 'c'
12

>>> lemma = 'walk'
>>> past_tense = lemma + 'ed'
>>> past_tense
'walked'
```

We can assign the name to a new value at any time:

```
>>> linguist_on_duty = 'Ann'
```

```
>>> linguist_on_duty
'Ann'                          The value of 'linguist_on_duty is currently 'Ann'
>>> linguist_on_duty = 'Sara'
>>> linguist_on_duty
'Sara'                         Now the value is 'Sara'
>>> linguist_on_duty = 'Ray'
>>> linguist_on_duty
'Ray'                          And now 'Ray'
```

When we assign a name to a new value we can have the name appearing on the right hand side as well as on the left:

```
>>> visitors = 1               The value of 'visitors' is 1
>>> visitors = visitors + 1    First evaluate the right hand side. The current
                               value of 'visitors' is 1;  1 + 1 is 2; so assign
                               2 to be the new value of visitors
>>> visitors
2
>>> visitors = visitors + 2    Evaluate the right hand side first. The current value
                               of visitors is 2;   2 + 2 is 4; so assign 4 to be the new
                               value
>>> visitors
4
>>>


>>> on_duty = 'Ann'
>>> on_duty
'Ann'
>>> on_duty = on_duty + ' and Sara'  The current value of on_duty is 'Ann' so
                                     on_duty + ' and Sara' becomes
                                     'Ann' + ' and Sara'
>>> on_duty
'Ann and Sara'
>>>
```

Let's look at one more example.

```
>>> staff = 'Ann'
>>> on_duty = staff
>>> staff = staff + ' and Sara'
```

What are the values of *staff* and *on_duty*?

```
>>> staff
'Ann and Sara'
>>> on_duty
'Ann'
```

2.2 Lists!

Let's look at another data type: lists. As the name suggests, a list is an ordered collection of objects. Lists are delimited by square brackets (**[ ]**) and the individual list elements are separated by commas.

Start the PythonWin Environment and type the following in the Interactive Window:

```
>>> my_test_scores = [83, 97, 90, 86]
```

As you can see, lists can contain numbers. They can also contain strings:

```
>>> linguists = ['Ann', 'Ben', 'Clara', 'Polly', 'Sally']
>>> computerists = ['Jake', 'Kim', 'Bill', 'Lori']
```

or both:

```
>>> stuff = ['the', 14, 'an', 27]
```

A list can even contain other lists:

```
>>> sentence = [ ['the', 'dog'], 'saw', ['the', 'cat']]
```

Here the sentence list contains three elements:

`['the', 'dog']`	the first element is a list containing two elements
`'saw'`	the second element is a string
`['the', 'cat']`	the third element is a list containing two elements

A list can also be empty:

```
>>> to_do_list = []
```

2.2.1 Accessing elements in a list

For some inexplicable reason, computer science types like to start counting from zero (*zero, one, two…)*. The person who developed Python (Guido van Rossum) is no exception. So, the index of the first element of a list is not '1', as you might expect, but '0'. To access an element in a list you use square brackets after the name:

```
>>> linguists[0]                the 0th element of the list linguists
'Ann'
>>> linguists[2]                the 2nd element of the list linguists
'Clara'
>>> sentence[0]
['the', 'dog']
```

(Here the first element of the sentence list is itself a list, ['the', 'dog'])

You can also use the same square bracket operator to get sublists:

```
>>> linguists[1:4]              linguists one through three
['Ben', 'Clara', 'Polly']
```

Again, for mostly inexplicable reasons, computer science types don't like counting the last number in a range. So [1:4] gives the 1st, 2nd and 3rd elements, but not the 4th – because the upper bound is not included.

If there is no number on the right side of the colon the range goes to the end of the list:

```
>>> linguists[1:]               linguists one to the end
['Ben', 'Clara', 'Polly', 'Sally']
```

If there is no number on the left side of the colon the range starts at the beginning of the list:

```
>>> linguists[:2]                linguists from the beginning to two
['Ann', 'Ben']
```

You can see all the elements of list by using the name of the list:

```
>>> my_test_scores
[83, 97, 90, 86]
```

What happens if you use an index that is larger than the number of items in the list?

```
>>> linguists[10]
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
IndexError: list index out of range
```

Python complains that the index is out of range (the index is larger than the size of the list).

2.2.2 Checking if something is a member of a list

Is Polly on the linguists list?

You can find out whether an item is in a list by using the **in** function:

```
>>> 'Polly' in linguists         Is 'Polly' in the list linguists?
True
>>> 'Steve' in linguists
False
```

('True' is Python's way of saying 'yes' and 'False' is its way of saying 'no')

2.2.3 The length of a list

How Many linguists are there?

You can find out how many elements there are in a list by using the length function, len:

```
>>> len([1, 2, 3, 4, 5])
5
>>> len(linguists)
5
>>> len(computerists)
4
```

2.2.4 Concatenating lists

You can concatenate lists (join two lists) by using the '+' operator:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> project_members = linguists + computerists
>>> project_members
['Ann', 'Ben', 'Clara', 'Polly', 'Sally', 'Jake', 'Kim', 'Bill', 'Lori']

>>> len(project_members)
9
```

2.2.5 A review of lists

So that's the quick 3-page overview of Python lists. Here's a little review.
You can either do this by typing the statements in PythonWin or by figuring this out in your head.

Let's say you have the following lists:

```
>>> linguists = ['Ann', 'Ben', 'Clara', 'Polly', 'Sally']
>>> computerists = ['Jake', 'Kim', 'Bill', 'Lori']
>>> classes =  ['syntax', 'morphology', 'pragmatics']
```

What's the answer to (you may want to cover the answers with a piece of paper):

	Answers
`classes[1]`	`'morphology'`
`a = classes[1:]`	`a has the value ['morphology', 'pragmatics']`
`a[1]`	`'pragmatics'`
`linguists[2]`	`'Clara'`
`classes = classes + ['phonology']`	`classes has the value ['syntax', 'morphology', 'pragmatics', 'phonology']`
`len(classes)`	`4`
`len(a)`	`2`
`'morphology' in classes`	`True`
`'morphology' in linguists`	`False`
`'morphology' in []`	`False`

What would you write in order to:

	Answers
assign the name 'head' to the first element in the computerists list.	`head = computerists[0]`
assign the name 'tail' to the computerists list minus the first element	`tail = computerists[1:]`
find the length of the tail list	`len(tail)`
add 'Ann' and 'Ben' to the beginning of the tail list	`tail = ['Ann', 'Ben'] + tail`
make a new **list** that contains the first element of computerists	`computerists[:1]` `(note computerists[1] does not return a list—that's why we use computerists[:1])`
make a new list that contains everything but the first 2 elements of computerists	`computerists[2:]`
make a new list, computerists2, that contains	`computerists2 = computerists[:1] +`

everything in the computerists list except for the second element.	`computerists[2:]`
find out if 'Ann' is in the computerists list	`'Ann' in computerists`

2.2.6 Other things you can do with lists

In the first tutorial I introduced you to functions and said that a rough analogy can be made to verbs and subcategorization—that is, a function can take arguments. So, for example, len takes one argument and returns a number representing the length of the argument:

```
>>> len(project_members)
9
>>> len('Ann Marie von Vanderviddle')
26
```

Methods are a special type of function that is 'attached' to some object. That is, a class of Python objects may have a set of functions that are associated with it.
Here's a weird analogy. Suppose I have two dogs, Kirby and Watson, that attended obedience school. They are so well trained that they only respond to commands issued to them and not the other dog. I issue these commands by first saying the dog's name and then the command:

Kirby, sit.
Watson, here.
Watson, down.
Kirby, here.

To make these look like Python method syntax, which uses a dot-notation, the commands would be

Kirby.sit()
Watson.here()
Watson.down()
Kirby.here()

So, looking at the first command (Kirby.sit()) I first reference the Python object—Kirby, in this case. Then I issue the command sit() and assume that the Python object (Kirby) has some internal set of instructions that tell it how to execute the command. It could be that Kirby and Watson execute the commands differently. For example, Watson might interpret 'here' as meaning to come to me, circle in back of me, come up on my left side, and sit. Kirby might interpret 'here' as coming to me and stand right in front of me.

In a similar way, I can issue commands to Python lists. Python does not have built in methods for 'here', 'down', and 'sit' but it does have the following useful methods associated with lists.

append

The append method adds an element to a list. Let's say I want to add 'Christine' to my list, linguists. I do the following:

```
>>> linguists.append('Christine')
```

Now let's look at the elements of the linguists list:

```
>>> linguists
['Ann', 'Ben', 'Clara', 'Polly', 'Sally', 'Christine']
```

'Christine' has been added to the list linguists.

You probably noticed that when you got to the period when you were typing 'linguists.' PythonWin displayed a popup menu. This menu contains all the methods applicable to lists. This is one of the great features of PythonWin—when you type in an object name followed by a period, it will display all the methods applicable to that object.

sort

You may have noticed the sort method in the menu. Let's give that a try.

```
>>> linguists.sort()
>>> linguists
['Ann', 'Ben', 'Christine', 'Clara', 'Polly', 'Sally']
```

Wow! That's easy. It puts the list in alphabetical order.

pop

Finally, let's take a look at one other method, pop. The pop method removes an element from the list and returns it:

```
>>> linguists.pop()
Sally
>>> linguists
['Ann', 'Ben', 'Christine', 'Clara', 'Polly']
```

As you will see throughout this set of tutorials, append and pop are very useful methods. Let's spend a bit of time playing with them.

2.2.7 To-do list example

Let's say I get to work in the morning, sit down, fire up PythonWin, and enter the following:

```
>>> to_do_list = []
```

So I start the day with an empty to-do list. How do I add the to-do item "email Ann" to the list? Just like this:

```
>>> to_do_list.append('email Ann')
```

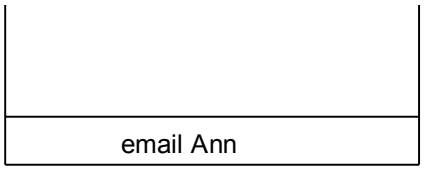
Let's add another to-do:

```
>>> to_do_list.append('read Kickapoo Adverbials article')
```

As you can see, I'm stacking up to-do items. Graphically, my to-do list started out being empty:

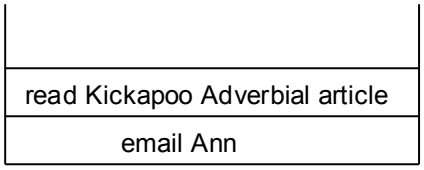
to-do-list

then I added the *email Ann* item:



to-do-list

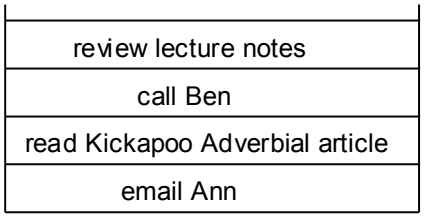
then I added the *read Kickapoo Adverbial article* to-do:



to-do-list

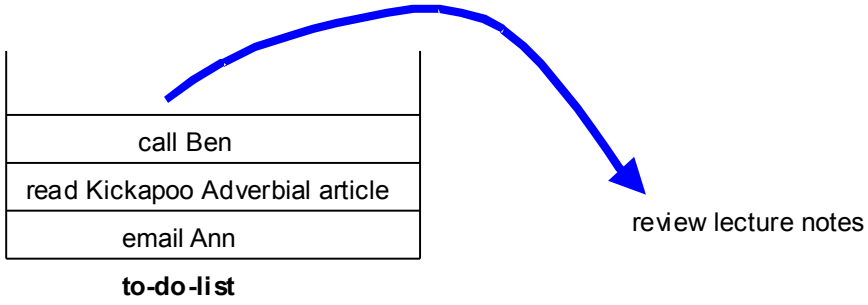
Let's add a few more:

```
>>> to_do_list.append('call Ben')
>>> to_do_list.append('review lecture notes')
>>> to_do_list
['email Ann', 'read Kickapoo Adverbial Article', 'call Ben', 'review lecture
notes']
```



to-do-list

This type of data structure is called a stack since it behaves like a stack of plates or blocks. New items are placed on the top of the stack. When we remove an item using pop, it's the top item we remove:




```
>>> to_do_list.pop()
'review lecture notes'

>>> to_do_list
['email Ann', 'read Kickapoo Adverbial Article', 'call Ben']

>>> to_do_list.pop()
'call Ben'

>>> to_do_list
['email Ann', 'read Kickapoo Adverbial Article']
```

2.3 Modules

As we've seen, Python has built-in functions and methods. For example, Python has the function len that will return the length of a list or string:

```
>>> len(['Ann', 'Ben'])
2
```

In addition, Python has many more functions and methods organized into modules. When you write a program you can import just those modules you need. We will be looking at modules in greater detail in future tutorials. Right now I'll give you a quick look at modules by presenting one example. Let's say I have a list of nouns:

```
>>> n = ['dog', 'cat', 'poodle', 'manx']
```

and let's say I want to select a random noun from the list. Python doesn't have a built-in function for this, but I can import a Python module called random. To do this I use the import statement:

```
>>> import random
```

This module has a function called 'choice' that randomly selects an item from a list. Here's an example:

```
>>> random.choice(n)
poodle
```

as you can see, the syntax is similar to that of methods. We have the module name, then a period followed by the name of the function. Finally, the arguments are enclosed in parentheses:

```
module_name.function_name(arguments)
```

Every time we call random.choice, a new item is randomly selected from the list:

```
>>> random.choice(n)
'manx'
>>> random.choice(n)
'dog'
>>> random.choice(n)
'poodle'
>>> random.choice(n)
'dog'
>>> random.choice(n)
'cat'
```

2.4 Defining your own functions

One of the powerful ideas behind programming languages is the ability to create your own functions. In Python we create a function by using the def form, which has the following syntax:

```
def new_function_name(arguments):
    """documentation string"""
    function body
```

For example:

```
def say_hello(name):
    """Print hello string"""
    print 'Hello, ' + name + '.'
    print 'Nice to meet you.'
```

Let's take a look at this line by line:

To say_hello	`def say_hello (name):`	We are instructing Python to construct a new function named 'say_hello'. The function takes one argument, which we have named 'name'.
	`"""Print hello string"""`	The optional documentation string (but I highly recommend that all you functions have this). It will be a short description of what the function does. In this case "Print hello string"
		To define a function you need to provide Python with a set of instructions.
First, print 'Hello' followed by the value of the name argument followed by a period	`print 'Hello, ' + name + '.'`	Instruction 1: print 'hello' + name
Then print 'Nice to	`Print 'Nice to meet you.'`	Instruction 2: print 'nice to meet you'

meet you.'		

Python is sensitive to spaces

Python expects things to line up. Notice how all the lines under the def line, are aligned? If, instead, the lines looked like this:

```
def say_hello(name):
     "Print hello string "
  print 'Hello, ' + name + '.'
            print 'Nice to meet you. '
```

Python would complain and you would get the following displayed on the bottom of the PythonWin window:

Failed to run script – syntax error – invalid syntax

Luckily, PythonWin is very good at automatically indenting lines. The general rule of thumb is that you should indent the block of lines that follow a line ending in a semicolon. In this case the def line contains a semicolon and you should indent the block of lines that describe that new function.

If you type functions in the PythonWin Interactive Window, Python will change the prompt from '>>>' to ' …' starting at the second line of the function, indicating that it is expected an indented block of lines. To terminate the definition of a function just enter a blank line.

```
>>> def say_hello (name):
...     """Print greeting"""
...     print 'Hello, ' + name + '.'
...     print 'Nice to meet you.'
...
```

to call (or 'run') your new function, type in the function name and the argument:

```
>>> say_hello('Ann')
Hello, Ann.
Nice to meet you.
```

When we call the function, the argument named 'name' is given the value 'Ann'. Thus, when the function gets to the line

```
print 'Hello, ' + name + '.'
```

'Hello, Ann.' is printed. (Remember that the '+' concatenates two strings)

The function we just created, say_hello, takes one argument, 'name'. However, we can write functions that take any number of arguments.

2.4.1 No arguments

It's easy to write a function that takes no arguments. The argument list () is empty:

```
>>> def hello():
...   """Print simple greeting"""
...   print 'Hello! Nice to meet you'
...
```

2.4.2 Multiple arguments

Multiple arguments (like the *name*, *likes*, and *friend* arguments in the example below) are separated by commas:

```
>>> def big_hello (name, likes, friend):
...     """Print complex greeting"""
...     print 'Hello, ' + name + '. '
...     print 'I hear you like ' + likes + '. '
...     print "And your friend's name is " + friend + '.'
...
```

You may have noticed that I commonly use single quotes to delimit strings but in the line

```
...   print "And your friend's name is " + friend + '.'
```

I used double quotes. Why did I do that?[1]

I can call this function this way:

```
>>> big_hello('Ann', 'Python', 'Clara')
Hello, Ann.
I hear you like Python.
And your friend's name is Clara.
>>> big_hello('Pat', 'poodles', 'Lori')
Hello, Pat.
I hear you like poodles.
And your friend's name is Lori.
```

2.4.3 Returning a value versus printing

Can you write a function that takes a string as an argument and prints out the length of the string (using a print statement)?

It might start with

```
>>> def my_length (word):
```

I wrote the function this way:

```
>>> def my_length(word):
...     print len(word)
...
```

Let's try it out and compare it to just using the function len:

```
>>> my_length('this')
4
>>> len('this')
4
>>>
```

[1] I used double quotes because the string I wanted to display contained a single quote in the word *friend's.* Alternatively, I could use the line 'And your friend\'s name is '

It looks like they have the same behavior but in fact there is a crucial difference between these two functions. Let's continue with our experiment:

```
>>> len('this') + len('cat')
7
```

The function len returns a number, which can be used for addition. When we try to do the same thing with my_length:

```
>>> my_length('this') + my_length('cat')
4
3
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
TypeError: unsupported operand types for +
```

we get a horrible error! Let's examine this in a bit more detail. What should happen as a result of calling my_length('this')? (Take a moment and look at how we defined my_length.) Well, the length of the string 'this' should be printed—that is, a '4' should be printed. If you look at the lines above, you will see that a '4' was indeed printed. Similarly my_length('cat') should print a three, and a '3' was printed. Our function appears to work perfectly, but the addition fails. This is caused by the difference between **returning** a value and **printing** a value. The function len **returns** a value, which I'll loosely describe as follows. Consider:

```
>>> len('this') + len('cat')
```

When `len('this')` is evaluated it returns a 4, and in a way we are substituting the function call `len('this')` with its value, 4:

```
>>> len('this') + len('cat')
        |
>>>     4       + len('cat')
```

When `len('cat')` is evaluated, a 3 is substituted for it:

```
>>> len('this') + len('cat')
        |            |
>>>     4       + len('cat')
                     |
>>>     4       +    3
```

Finally, Python can perform the simple addition and return the answer, 7.
Since len returns a value it can be used on the right side of an assignment:

```
>>> a = len('this')
```

Notice that len doesn't contain a print statement, as we don't see '4' printed to the screen. Now, let's try it with my_length:

```
>>> b = my_length('this')
4
```

This time we did see something printed to the screen.
What's the value of 'a'?

```
>>> a
4
```

This is what we expected. len('this') returns the value, 4, and we then have a simple assignment statement:

```
>>> a = 4
```

What's the value of 'b'?

```
>>> b
>>>
```

Hmmm. it doesn't seem that 'b' has a value. That's because my_length doesn't return any value.[2] All it does is print. We can write functions that return values by using the return function:

```
>>> def my_new_length(word):
...     return len(word)                    return the value of  len(word)
...
```

Let's give this a try:

```
>>> my_new_length('this') + my_new_length('cat')
7
>>>
```

Now it works fine! So what's the moral? Am I saying that you should not use print statements in your own functions? Nope. That's not it at all. There are plenty of times where a print statement is exactly what you need. However, if the results of your function are to be used by some other part of your script, you need to be **returning** the results not **printing** them.

Can you write a function 'add' that takes two arguments and returns their sum? It should behave as follows:

```
>>> add(2, 5)
7
>>> add(2, 5) * add(2,3)
35
```

For an example of one way to write this function, see the solutions at the end of this tutorial.

Let's go through another example. Suppose you are a runner and you want to write a function that will convert distances measured in kilometers to miles. The formula is

miles = kilometers * .621

and, as an experiment, we write the following two functions:

```
>>> def km2miles(km):
...     print km * .621
...
>>> def km2miles2(km):
...     return km  * .621
```

[2] I will talk more about this in a later tutorial.

```
...
```

Now let's try them out:

```
>>> km2miles(10)
6.21
>>> km2miles2(10)
6.21
```

Both these functions encode the idea that to convert from kilometers to miles you multiply the kilometers by . 621.
They both seem to work equally well. If this is the only way we will use the functions then either could be used. But let's say that you've entered a duathalon, which has a 10k run, a 26 mile bike, followed by another 10k run, and you want to find the total distance. km2Miles2, the function that uses return, works perfectly:

```
>>> km2miles2(10) * 2 + 26
38.42
```

but km2Miles, the function that uses print and no return, generates an error.

2.5 A sentence generator

Throughout most of this tutorial you have been typing examples in PythonWin's interactive window. Let's quickly review how to create a new file containing a Python program. You click on the 'new' icon (the blank piece of paper) in the PythonWin environment. A popup window will appear giving you a choice of *Python Script* or *Grep.* Select *Python Script* and press OK. Now type in the following in the new window:

```
import random
n = ['dog', 'cat', 'poodle', 'frog']
print random.choice(n)
```

Now save the file by pressing the save icon (the floppy disk) and typing in a filename (I used the name 'sentence_generator'). Run the program by pressing the 'run' icon. When I did this, the word 'dog' was printed in the Interactive Window, but since we are randomly selecting an element from the list you result may be different.

Let's say I want to create a function, noun, which will give me the following results:

```
>>> noun()
'frog'
>>> noun()
'cat'
>>> noun()
'dog'
>>>
```

That is, I want to create a function that will return a random noun. I'll revise my sentence_generator.py file to look like:

```
import random
def noun():
    """Return a random noun"""
```

```
    n = ['dog', 'cat', 'poodle', 'frog']
    return random.choice(n)
```

When I now run this program the function, noun, is defined. I can then use this function within the interactive window. That is, I can type:

```
 >>> noun()
'frog'
```

and get a noun returned (in this case *frog*).

Let me add a few more functions to the file:

```
def det():
    """Return a random determiner"""
    return random.choice(['a', 'the'])

def adj():
    """Return a random adjective"""
    return random.choice(['sad', 'happy', 'silly'])
```

When I run the file, these new functions are defined. Let's test them:

```
>>> adj()
'sad'
>>> adj()
'happy'
>>> det()
'the'
>>> det()
'a'
```

Notice that in my noun function I had two lines:

```
    n = ['dog', 'cat', 'poodle', 'frog']
    return random.choice(n)
```

but in my adj and det functions I condensed this to one line:

```
    return random.choice(['sad', 'happy', 'silly'])
```

They behave the same way and the choice of which to use is up to you. In future tutorials I will explain why you might choose one approach over the other.

Remember that we can create longer strings by concatenating to strings like this:

```
>>> 'the' + ' ' + 'dog'
'the dog'
```

Using the '+' operator I concatenate three strings ('the', ' ', and 'dog') to create the string 'the dog'.

How can I create a string representing a noun phrase consisting of a determiner, adjective, and noun, by using the functions we just defined?

If you think you know the answer, try it out by using the interactive window before looking at the answer in this footnote.[3]

Now, try your hand at writing the function, np, which will have the following behavior:

```
>>> np()
'a silly frog'
>>> np()
'the happy frog'
>>> np()
'a sad poodle'
>>> np()
'a silly poodle'
```

How did you do? Here's one way of doing it:

```
def np():
    """Returns a random noun phrase"""
    return det() + ' ' + adj() + ' ' + noun()
```

This definition represents the following context free rule:

NP → det adj noun

'An NP consists of a determiner followed by an adjective followed by a noun'

Exercise 2.1:

You job is to continue writing this little grammar. Here's a set of context free rules to get you started:

NP → det adj noun
VP → verb NP
S → NP VP

lexicon:

adjectives: *sad, happy, silly*
determiners: *the a*
nouns: *cat, dog, poodle, frog*
verbs: *likes, saw*

Your program should produce something like the following:

```
>>> s()
'the happy dog saw a silly poodle'
>>> s()
'a happy poodle likes a sad poodle'
>>> s()
'the sad poodle likes a silly cat'
>>> s()
'a sad dog likes a happy frog'
>>>
```

[3] `det() + ' ' + adj() + ' ' + noun()`

2.5.1 Enhancements:

Instead of 'the sad poodle likes a silly cat' it would be nice to capitalize the first word of the sentence and put a period at the end. You can capitalize the first letter of a string by using the capitalize() method. For example,

```
>>> a = 'bush'
>>> a.capitalize()
'Bush'
>>> noun().capitalize()
'Poodle'
```

In this last example (`noun().capitalize()`) `noun()` returns a string and then the method 'capitalize' applies to this string.

Can you alter your program to return a sentence with the first letter capitalized and with a final period?

For one solution to this exercise see the solutions at the end of this tutorial or you can get the solution as a python file from http://www.zacharski.org/python/

2.5.2 Another Revision.

You may have noticed that the functions for generating nouns, verbs, adjectives, and determiners are quite similar. We can reduce these four functions to one more generic one. Suppose we have the following lists defined:

```
nouns =       ['dog', 'cat', 'poodle', 'frog']
verbs =       ['likes', 'saw']
adjectives =  ['sad', 'happy', 'silly']
determiners = ['a', 'the']
```

Can you create a generic function that takes one of these lists as an argument and returns a random element from it? For example,

```
>>> select_lexical_item(nouns)
'poodle'
>>> select_lexical_item(adjectives)
'sad'
>>> select_lexical_item(verbs)
'likes'
```

Your answer to this will be used in the next tutorial, so take your time and work this through.[4]

[4]One solution is:

```
def select_lexical_item(list):
    return random.choice(list)
```

Comments

Once again, I am very interested in getting your comments. If things are unclear, if you have suggestions, or if you want some help in learning to program, please contact me at: ron@zacharski.org

Solutions

The add function:

```
>>> def add(x, y):
...     return x + y
```

2.1 The Sentence Generator

(This solution is also available as a file at http://www.zacharski.org/python)

```
import random

def noun():
    "Return a random noun"
    n = ['dog', 'cat', 'poodle', 'manx']
    return random.choice(n)

def det():
    "Return a random determiner"
    return random.choice(['a', 'the'])

def adj():
    "Return a random adjective"
    return random.choice(['sad', 'happy', 'silly'])

def verb():
    "Return a transitive verb"
    return random.choice(['likes', 'saw'])

def np():
    "Returns a random noun phrase"
    return det() + ' ' + adj() + ' ' + noun()

def vp():
    "Returns a random verb phrase"
    return verb() +  ' ' + np()

def s():
    "Return a simple sentence"
    return np().capitalize() + ' ' + vp() + '.'
```