> *I'm afraid I'm not personally qualified to confuse cats, but I can recommend an extremely good service.*

Mutable Objects

Mutable and Immutable

As we are about to see **list**s, are our first example of a mutable sequence and object, now is a good time to discuss **mutability**. Mutability is the description of whether an object's data can be modified.

A **mutable** object is an object whose data is **modifiable**. The data can be modified either through the methods of the object or through direct access to the data itself. An **immutable** object is an object where the data **cannot** be modified in anyway. Strings are examples of immutable objects. It is not possible to modify the data of a string. We are about to meet our first mutable objects: **lists**.

Lists

Lists are **mutable** sequences. We can access elements at a particular index as we did with strings and use slicing, but we can also modify a list by changing elements or by adding and removing elements.

Here are some examples of lists in action:

```
>>> a = [1,2,3,4]
>>> type(a)
<class 'list'>
>>> a[0]
1
>>> a[-1]
4
>>> a[1:3]
[2, 3]
>>> a[::2]
[1, 3]
>>> a[0] = 5
>>> a
[5, 2, 3, 4]
>>> a.append(7)
>>> a
[5, 2, 3, 4, 7]
>>> a.pop(3)
4
>>> a
[5, 2, 3, 7]
>>> a.insert(1,9)
>>> a
[5, 9, 2, 3, 7]
>>> a.sort()
>>> a
[2, 3, 5, 7, 9]
>>>
```

The first example shows the syntax for lists and the type, the next four examples look up information in the list using indexing and slicing. In the next example, we see indexing used on the left hand side of an assignment statement. The semantics is that the value stored at the supplied index is updated with the value on the right hand side of the assignment. The last four examples use methods of the list class to perform more operations on the list.

The first of these is the `append` method. The append method adds the item that is the argument to the end of the list. This is followed by the `pop` method. Pop removes the item at the given index from the list and returns that item. Another way to add items to the list is the `insert` method. Insert takes an index and an item as arguments and inserts the item at the given index, pushing the other items down the list. The last example shows how we can `sort` the elements — arrange them in order.

Notice how all the methods and operations performed on the list modify the list stored in the variable being used and most do not return anything. Compare this to strings, where all methods and operations return a new item and do not change the original string in the variable.

Python also includes a function for converting any sequence into a list of the objects in that sequence.

```
>>> list("spam")
['s', 'p', 'a', 'm']
>>> list((1, 7, 8, 42))
[1, 7, 8, 42]
```

Because lists are sequences we can iterate over a list using a for loop.

```
>>> for i in [0, 1, 2, 3, 4] :
    print(i)

0
1
2
3
4
```

A for loop is not much good if we do not do anything with the items. Here is an example of taking a list of numbers and generating a new list with all the numbers from the original list squared.

```
>>> x = [2, 5, 8, 14, 18]
>>> y = []
>>> for i in x :
        y.append(i ** 2)

>>> y
[4, 25, 64, 196, 324]
```

Range

Because iterating over sequences of numbers is very common, Python comes with a function for generating a sequence of numbers. This function is called range. range generates a special object that is iterable, meaning we can loop over the elements in the range using the for loop.

```
>>> for i in range(10) :
    print(i, end=" ")

0 1 2 3 4 5 6 7 8 9
```

In short, range creates an object that can be looped through that contains a sequence of numbers from a start number up to but not including a given end number. The end number is required and is excluded from the sequence as usually we use range to generate indices. The default start number is zero; this is also for generating indices as indexing starts at zero. The step size can also be changed, enabling us to skip numbers instead of going one number at a time. This can be seen in the following examples. Note, to simplify these examples the list function has been used.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 10, 3))
[1, 4, 7]
>>> list(range(10, 1 ,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

range Syntax

range(*start*, *end*, *step_size*)

start is optional and defaults to 0, *step_size* is optional and defaults to 1

Semantics

Generates an iterable sequence of numbers. If only `end` is given, then the sequence will start at 0 go up to but not including `end`. If `start` and `end` are given, then the sequence will start at `start` and go up to but not including `end`. If all three options are given then the sequence will start at `start` and go up to but not including `end` with a given `step_size`.

Pass by Reference

The following example is of a function that takes a list and a number and adds all numbers from 0 up to but not including that number to the list.

```
def add_to(existing_list, num) :
    """Adds numbers 0 to 'num' to the end of 'list'
    Parameters:
        existing_list (list): The list to which numbers will be added.
        num (int): The number up to which will be added to 'existing_list'.
    Examples:
        >>> a_list = []
        >>> add_to(a_list, 4)
        >>> a_list
        [0, 1, 2, 3]
    """
    for i in range(num) :
        existing_list.append(i)
```

After saving the file as add_to.py and running here is a test.

```
>>> a_list = [3, 5, 6, 87, 1, 5]
>>> add_to(a_list, 5)
>>> a_list
[3, 5, 6, 87, 1, 5, 0, 1, 2, 3, 4]
```

Notice how even though the function does not return anything the list is modified simply by passing it to the function. We say that objects in Python are **passed by reference**, which means that objects passed into functions are passed directly. Consequently, a mutable object, which is changed inside a function body, will still be changed after the function call. In contrast, some programming languages use a **pass by value** strategy, where a copy of the object is passed into functions, so any modifications will not affect the original object.

The `add_to` function is what is often called a **procedure**, rather than a function, because it does not return a value. In Python, functions or procedures always return a value, even if there is no `return` statement. Procedures, like this example, return `None` if there is no return line. `None` is an instance of a special type in Python called `NoneType` which has only one value, `None`.

```
>>> type(None)
<class 'NoneType'>
>>> a = None
>>> a
>>>
>>> help(None)
Help on NoneType object:

class NoneType(object)
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __repr__(self, /)
 |      Return repr(self).
```

`None` is treated specially by the interpreter. The second example shows that anything that evaluates to `None` is not printed to the interpreter.

> **Pass by Reference**
>
> If a list is passed into a function, **any changes** made to that list inside the function will affect the list **outside** of the function, because lists are mutable.