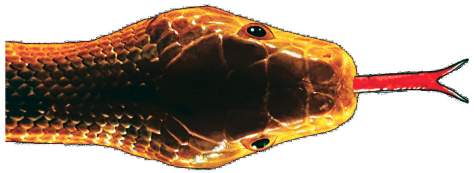
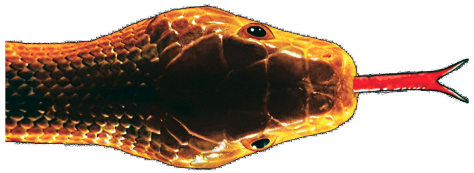
Object-Oriented Programming in Python
Goldwasser and Letscher
Chapter 2

Getting Started with Python



The Python Interpreter

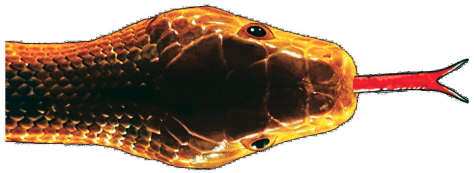
- A piece of software that executes commands for the Python language
- Start the interpreter by typing **python** at a command prompt
- Many developers use an Integrated Development Environment for Python known as IDLE



The Python Prompt

>>>

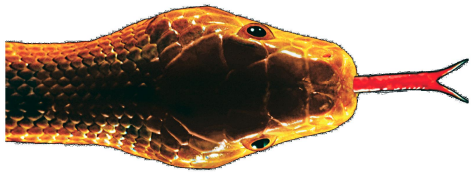
- This lets us know that the interpreter awaits our next command



Our First Example

```
>>> groceries = list()
>>>
```

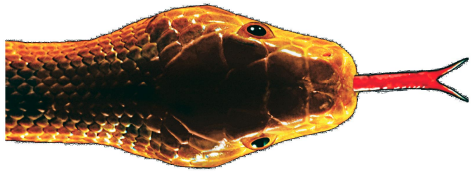
- We see a new Python prompt, so the command has completed.
- But what did it do?



Instantiation

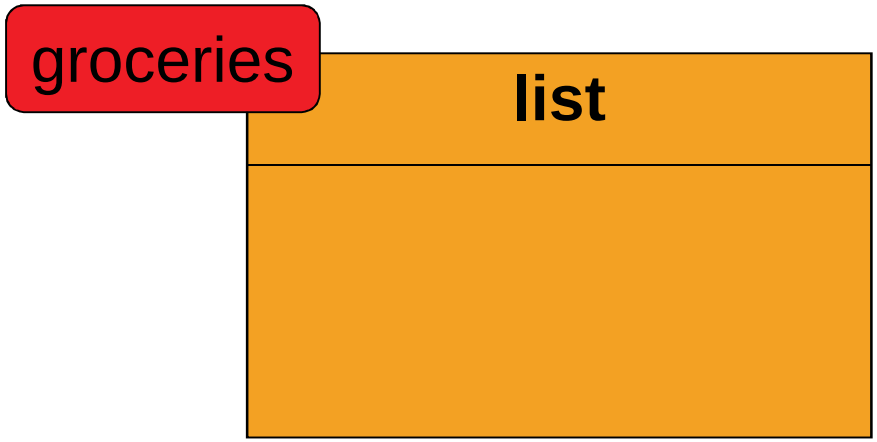
```
>>> groceries = list()
>>>
```

Constructs a new instance from the **`list`** class (notice the parentheses in the syntax)

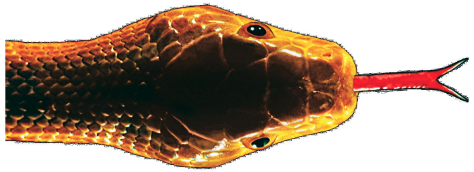


Assignment Statement

```
>>> groceries = list()
>>>
```

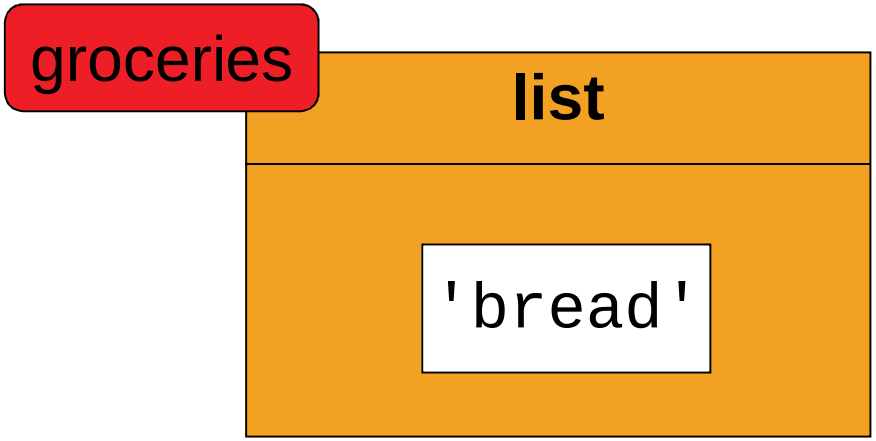


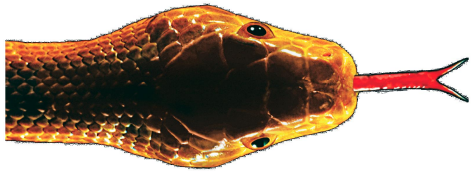
groceries serves as an **identifier** for the newly constructed object (like a “sticky label”)



Calling a Method

```
>>> groceries = list()
>>> groceries.append('bread')
>>>
```



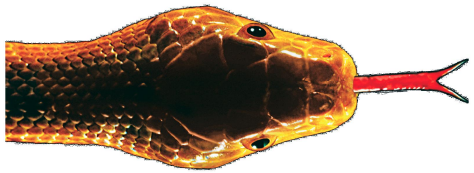


Displaying Internals

When working in the interpreter, we do not directly "see" the internal picture. But we can request a textual representation.

```
>>> groceries = list()
>>> groceries.append('bread')
>>> groceries
['bread']
>>>
```

⇐ interpreter's response

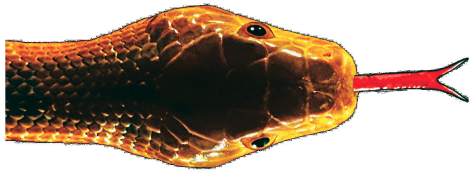


Method Calling Syntax

```
groceries.append('bread')
```

object method parameters

- There may be many objects to choose from
- The given object may support many methods
- Use of parameters depends upon the method



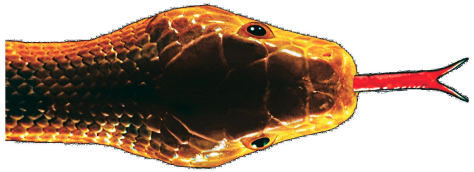
Common Errors

```
>>> groceries.append()
Traceback (most recent call last):
  File "<stdin>", line 1, in -toplevel-
TypeError: append() takes exactly one argument (0 given)
```

What's the mistake?

```
>>> groceries.append(bread)
Traceback (most recent call last):
  File "<stdin>", line 1, in -toplevel-
NameError: name 'bread' is not defined
```

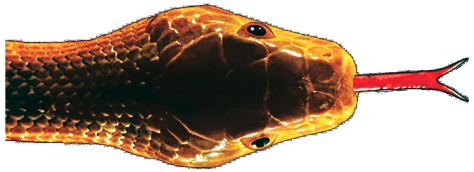
What's the mistake?



The append Method

New item added to the **end** of the list
(much like a restaurant's waitlist)

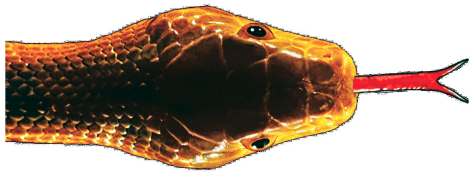
```
>>> waitlist = list()
>>> waitlist.append('Kim')
>>> waitlist.append('Eric')
>>> waitlist.append('Nell')
>>> waitlist
['Kim', 'Eric', 'Nell']
```



The insert Method

Can insert an item in an arbitrary place using a numeric **index** to describe the position. An element's index is the number of items **before** it.

```
>>> waitlist
['Kim', 'Eric', 'Nell']
>>> waitlist.insert(1, 'Donald')
>>> waitlist
['Kim', 'Donald', 'Eric', 'Nell']
```

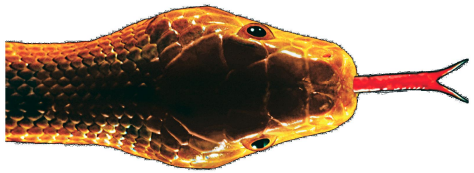


Zero-Indexing

By this definition,

- the first element of the list has **index 0**
- the second element has **index 1**
- the last element has index **(length - 1)**

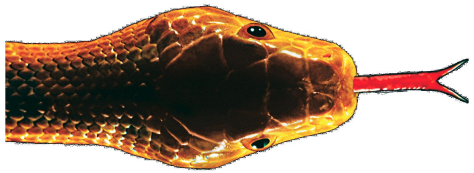
We call this convention **zero-indexing**.
(this is a common point of confusion)



The remove Method

What if Eric gets tired of waiting?

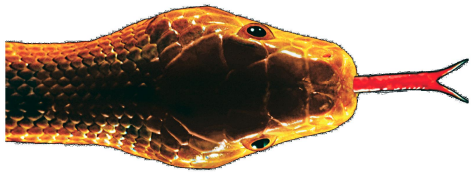
```
>>> waitlist
['Kim', 'Donald', 'Eric', 'Nell']
>>> waitlist.remove('Eric')
>>> waitlist
['Kim', 'Donald', 'Nell']
>>>
```



The remove Method

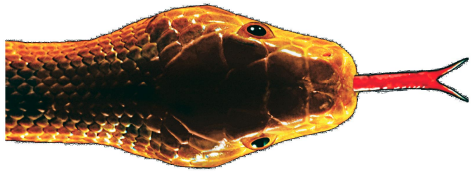
- Notice that we didn't have to identify where the item is; the list will find it.
- If it doesn't exist, a ValueError occurs
- With duplicates, the **earliest** is removed

```
>>> groceries
['milk', 'bread', 'cheese', 'bread']
>>> groceries.remove('bread')
>>> groceries
['milk', 'cheese', 'bread']
```



Return values

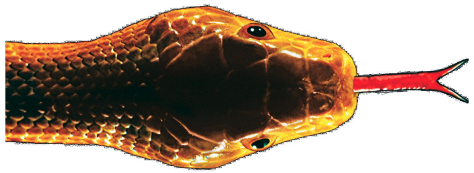
- Thus far, all of the methods we have seen have an effect on the list, but none return any direct information to us.
- Many other methods provide an explicit **return value**.
- As our first example: the count method



The count method

```
>>> groceries
['milk', 'bread', 'cheese', 'bread']
>>> groceries.count('bread')
2
>>> groceries.count('milk')
1
>>> groceries.count('apple')
0
>>>
```

⇐ response from the interpreter

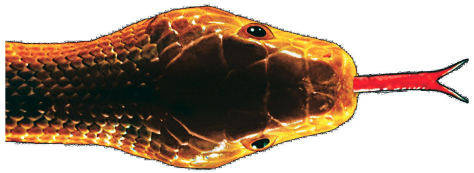


Saving a Return Value

- We can assign an identifier to the returned object

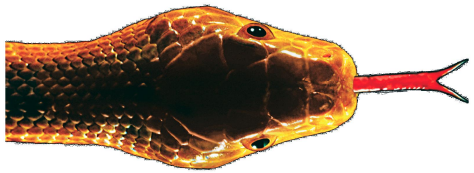
```
>>> groceries
['milk', 'bread', 'cheese', 'bread']
>>> numLoaves = groceries.count('bread')
>>> numLoaves
>>> 2
```

- Notice that it is no longer displayed by interpreter
- Yet we can use it in subsequent commands



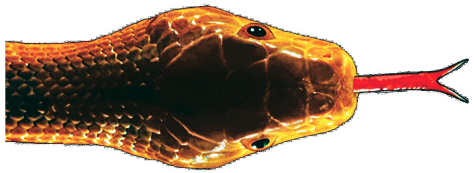
Operators

- Most behaviors are invoked with the typical "method calling" syntax of `object.method( )`
- But Python uses shorthand syntax for many of the most common behaviors (programmers don't like extra typing)
- For example, the length of a list can be queried as `len(groceries)` although this is really shorthand for a call `groceries.__len__( )`



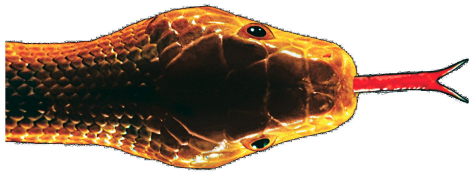
Accessing a list element

```
>>> waitlist
['Kim', 'Donald', 'Eric', 'Nell']
>>> waitlist[1]
'Donald'
>>> waitlist[3]
'Nell'
>>> waitlist[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```



Negative Indices

```
>>> waitlist
['Kim', 'Donald', 'Eric', 'Nell']
>>> waitlist[-1]
'Nell'
>>> waitlist[-3]
'Donald'
>>> waitlist[-4]
'Kim'
>>>
```



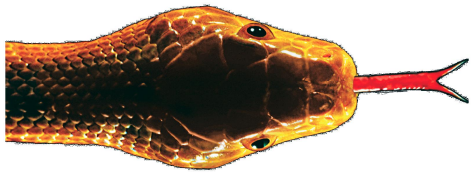
List Literals

- We originally used the syntax **list**( ) to create a new empty list. For convenience, there is a shorthand syntax known as a list literal.

  groceries = [ ]

  (experienced programmers like to type less!)
- List literals can also be used to create non-empty lists, using a syntax similar to the one the interpreter uses when displaying a list.

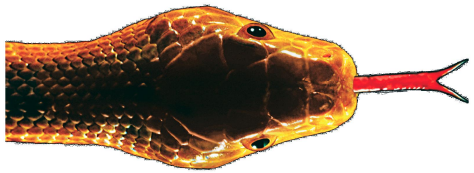
  groceries = `['cheese','bread','milk']`



Copying Lists

- The list( ) constructor is useful for making a new list modeled upon an existing sequence

```
>>> favoriteColors = ['red', 'green', 'purple', 'blue']
>>> primaryColors = list(favoriteColors)        ⇐ a copy
>>> primaryColors.remove('purple')
>>> primaryColors
['red', 'green', 'blue']
>>> favoriteColors
['red', 'green', 'purple', 'blue']
>>>
```



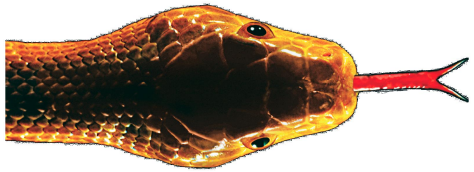
The range function

- Lists of integers are commonly needed. Python supports a built-in function named **range** to easily construct such lists.
- There are three basic forms:

  range(stop)

  goes from zero **up to but not including** stop

```
>>> range(5)
[0, 1, 2, 3, 4]
```



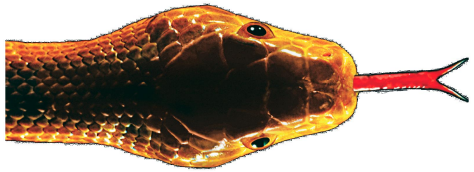
The range function

range(start, stop) begins with start rather than zero

```
>>> range(23, 28)
[23, 24, 25, 26, 27]
```

range(start, stop, step) uses the given step size

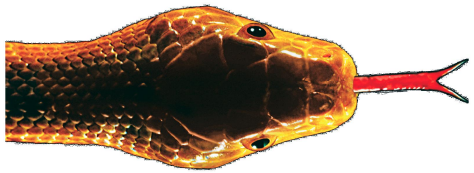
```
>>> range(23, 35, 4)
[23, 27, 31]
>>> range(8, 3, -1)
[8, 7, 6, 5, 4]
```



Many useful behaviors

• groceries.pop( )	remove last element
• groceries.pop(i)	remove i^{th} element
• groceries.reverse( )	reverse the list
• groceries.sort( )	sort the list
• 'milk' in groceries	does list contain?
• groceries.index('cereal')	find leftmost match

These will become familiar with more practice.



Documentation

- See Section 2.2.6 of the book for more details and a table summarizing the most commonly used list behaviors.

- You may also type help(list) from within the Python interpreter for documentation, or for a specific method as help(list.insert)