Introduction to Python and Pygame

Why Python?

- ***Python is easy to use***
  - *Python typically operates at a much higher level of abstraction.*
  - *Syntax rules are very simple.*
  - Python to take one-fifth the time it would if coded in C or Java

Why Python?

- ***Python is expressive***
- *Expressive in this context means that a single line* of Python code can do more than a single line of code in most other languages
- Example:
  - In java
    - int temp = var1;
    - var1 = var2;
    - var2 = temp;
  - Python
    - var2, var1 = var1, var2

Why Python?

- ***Python is readable***
  - ***one can guess easily what's happening in code***

```
#Perl version.
sub pairwise_sum {
my($arg1, $arg2) = @_;
my(@result) = ();
@list1 = @$arg1;
@list2 = @$arg2;
for($i=0; $i < length(@list1);
$i++) {
push(@result, $list1[$i] +
$list2[$i]);
}
return(\@result);
}
```

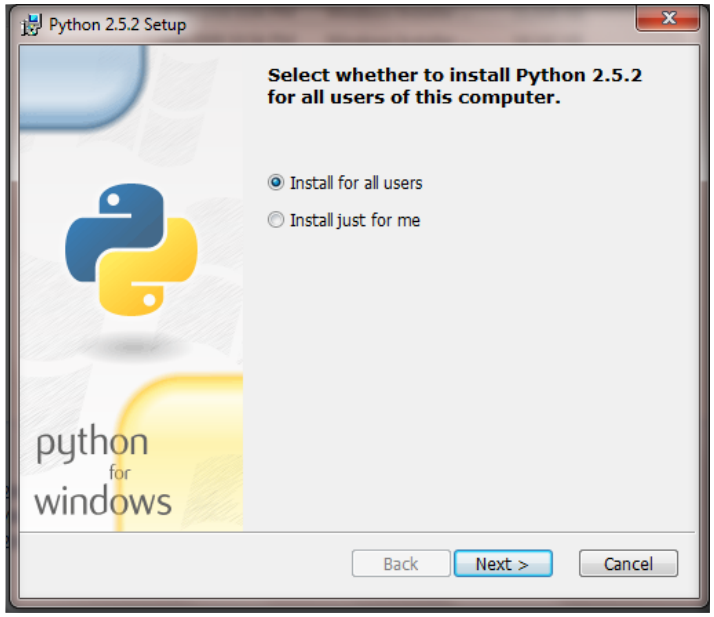
```
# Python version.
def pairwise_sum(list1, list2):
result = []
for i in range(len(list1)):
result.append(list1[i] + list2[i])
return result
```

Why Python?

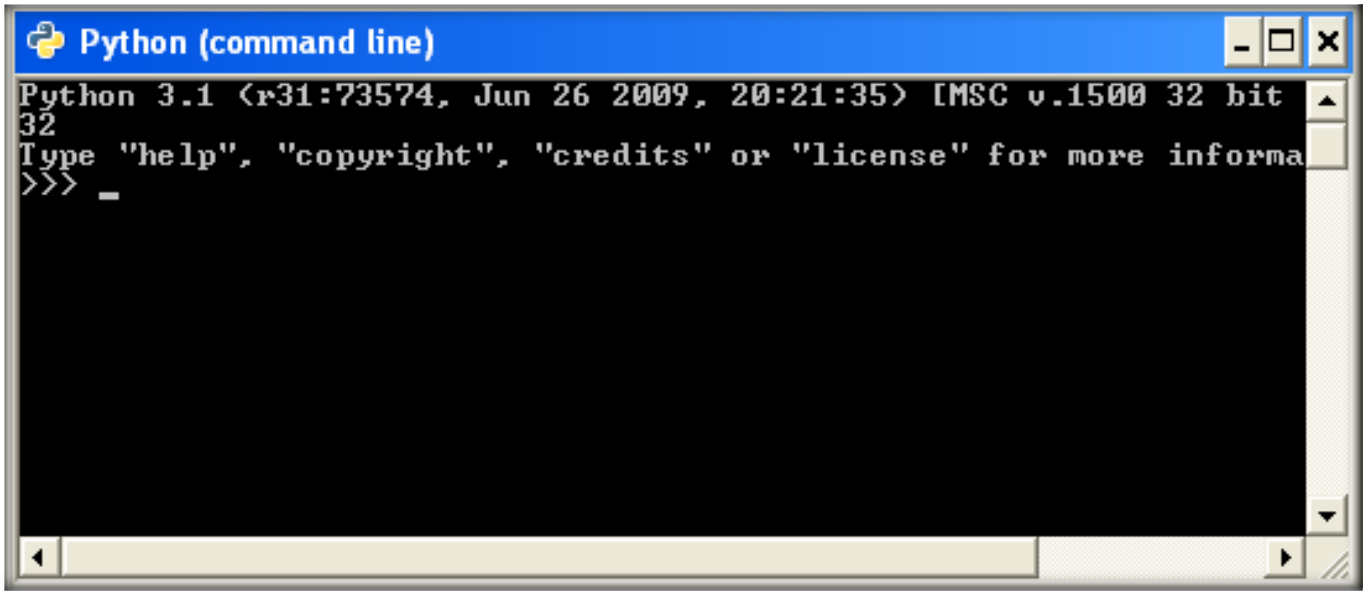
- ***Python is complete***
  - Python standard library comes with modules for handling email, web pages, databases, operating system calls, GUI development, and more.
- ***Python is cross-platform***
  - Python runs on many different platforms: Windows, Mac, Linux, UNIX, and so on.
- ***Python is free***
  - Python was originally, and continues to be, developed under the open source model, and it's freely available. You can download and install practically any version of Python and use it to develop software for commercial or personal applications, and you don't need to pay a dime.

Installing Python

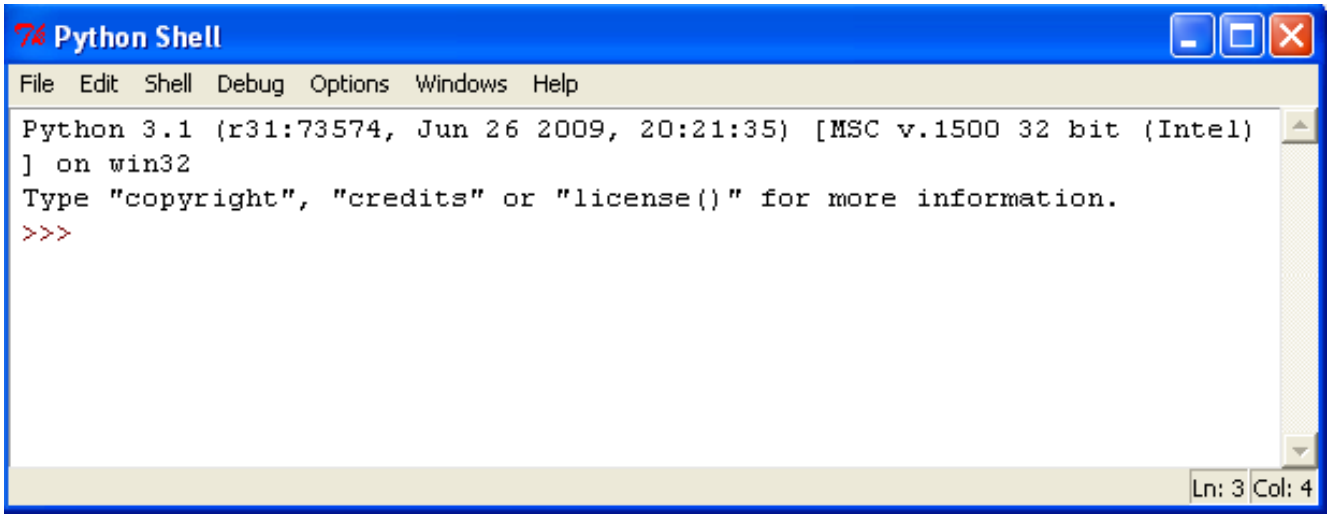
- Installing Python is a simple matter, regardless of which platform you're using. the most recent one can always be found at www.python.org.



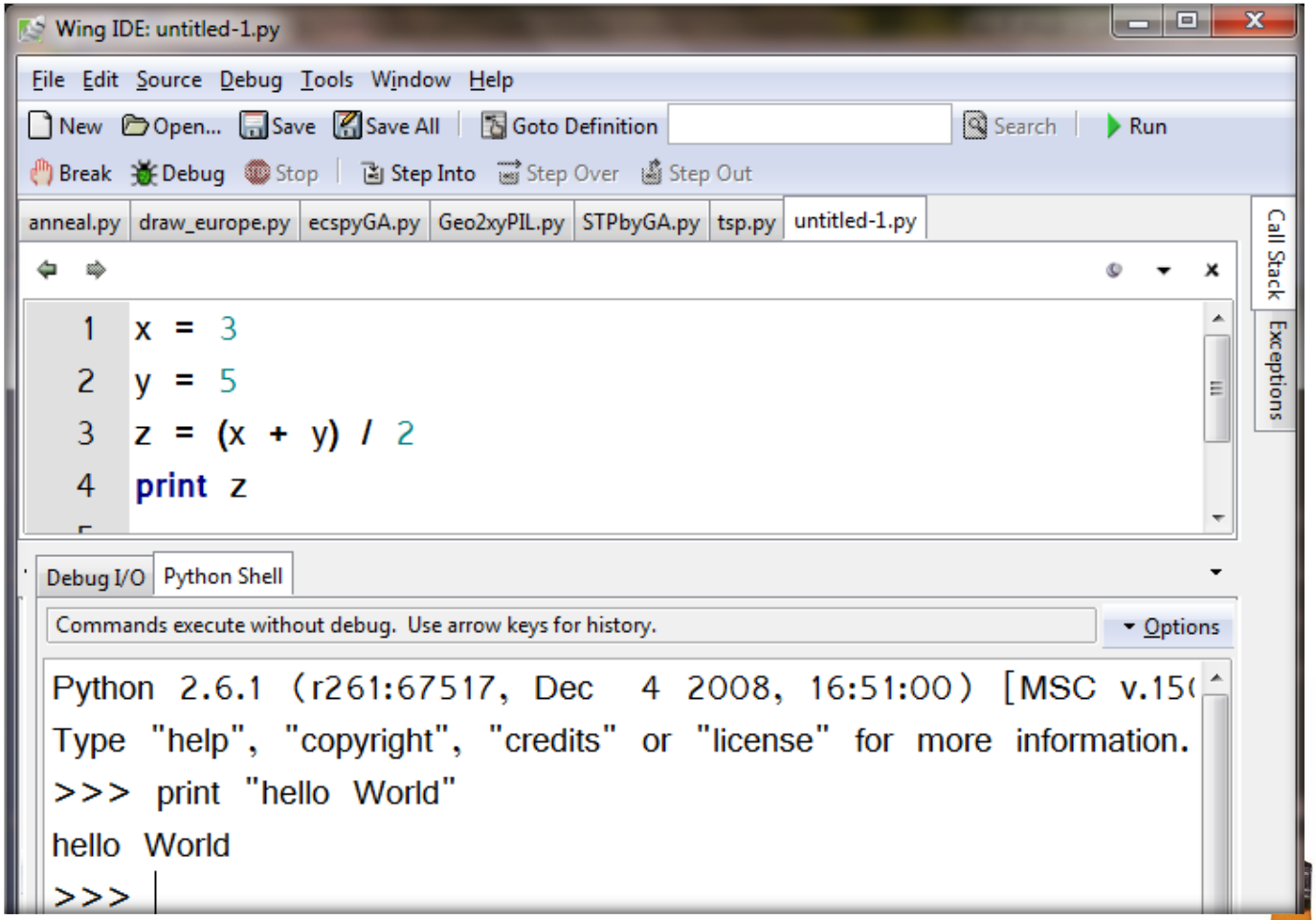
command-line



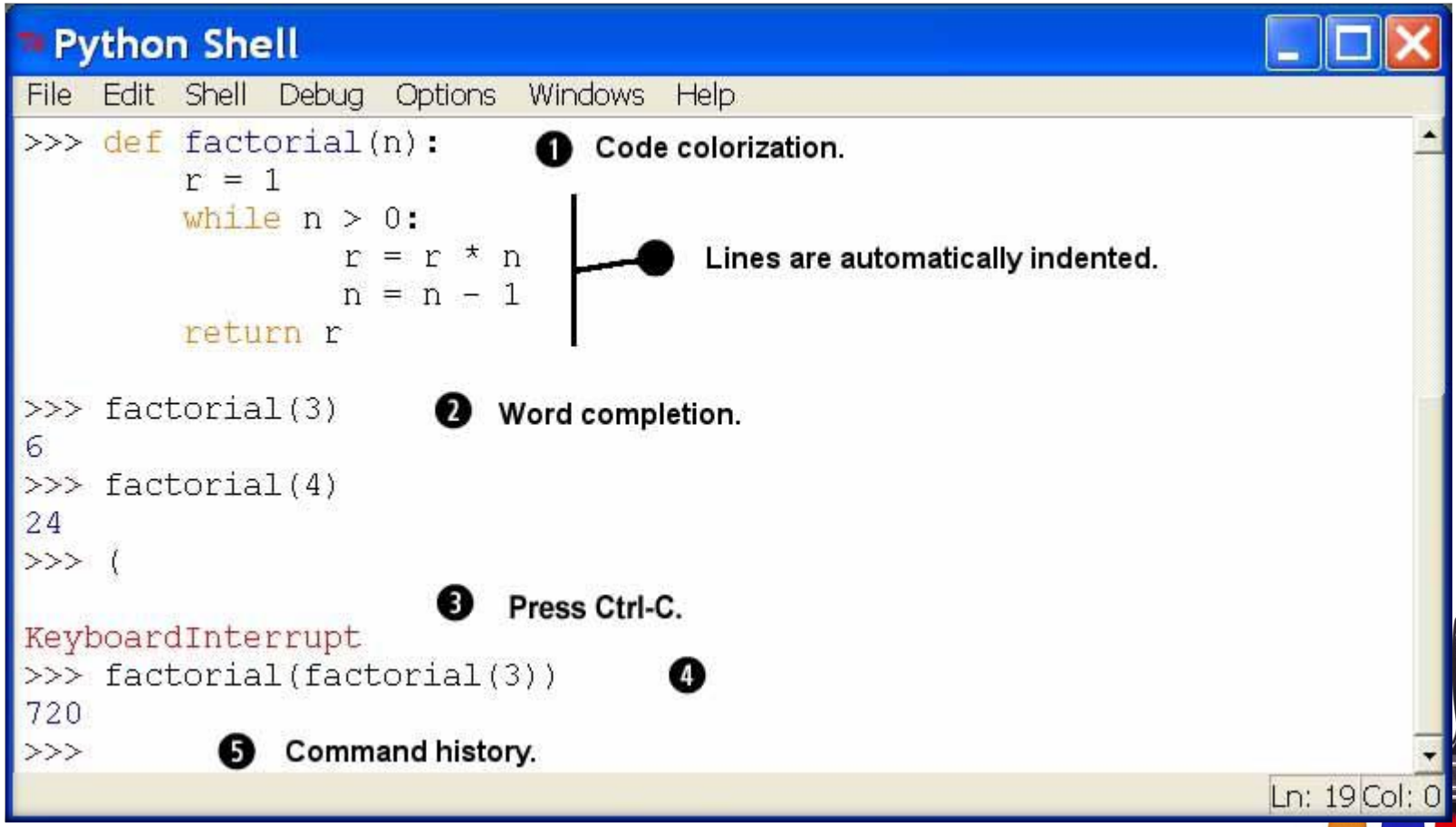
The IDLE integrated development environment



Interactive and Programing Mode



Using IDLE's Python Shell window



Comments

- For the most part, anything following a # symbol in a Python file is a comment and is disregarded by the language.

```
1  # Assign 5 to x
2  x = 5
3  x = 3 # Now x is 3
4  x = "# This is not a comment"
5  """ Multi Line Comments
6  Second Line"""
```

Variables and assignments

```
>>> x = "Hello"
>>> print(x)
Hello
>>> x = 5
>>> print(x)
5
```

del statement

- The del statement deletes the variable.

```
>>> x = 5
>>> print(x)
5
>>> del x
>>> print(x)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Expressions

- Python supports arithmetic and similar expressions; these will be familiar to most readers. The following code calculates the average of 3 and 5, leaving the result in the variable z:

```
1  x = 3
2  y = 5
3  z = (x + y) / 2
```

Strings

- You can use single quotes instead of double quotes. The following two lines do the same thing:
- x = "Hello, World"
- x = 'Hello, World'

Numbers

- Python offers four kinds of numbers: *integers, floats, complex numbers, and Booleans.*

```
>>> 5 + 2 - 3 * 2
1
>>> 5 / 2          # floating point result with normal division
2.5
>>> 5 / 2.0        # also a floating point result
2.5
>>> 5 // 2         # integer result with truncation when divided using '//'
2
>>> 30000000000    # This would be too large to be an int in many languages
30000000000
>>> 30000000000 * 3
90000000000
>>> 30000000000 * 3.0
90000000000.0
```

Numbers

```
>>> 2.0e-8          # Scientific notation gives back a float
2e-08
>>> 3000000 * 3000000
9000000000000
>>> int(200.2)
200
>>> int(2e2)
200
>>> float(200)
200.0
```

These are explicit conversions between types 1. int will truncate float values.

Built-in numeric functions

- Python provides the following number-related functions as part of its core:
- abs, divmod, cmp, coerce, float, hex, int, long, max, min, oct, pow, round

```
Python 2.6.1 (r261:67517, Dec
Type "help", "copyright", "credits"
>>> abs(-2)
2
>>> divmod(5,2)
(2, 1)
>>> max(4,2)
4
>>> round(4.313)
4.0
```

Getting input from the user

- You can also use the input() function to get input from the user. Use the prompt string you want displayed to the user as input's parameter:

```
>>> name = input("Name? ")
Name? Vern
>>> print(name)
Vern
>>> age = int(input("Age? "))
Age? 28
>>> print(age)
28
>>>
```

Converts input from string to int

Lists

- ***Lists are like arrays***
- A list in Python is much the same thing as an array in Java or C or any other language. It's an ordered collection of objects. You create a listd by enclosing a comma separated list of elements in square brackets, like so:
- Example
  - x = [1, 2, 3]

List indices

Elements can be extracted from a Python list using a notation like C's array indexing.

Like C and many other languages, Python starts counting from 0; asking for element 0

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[2]
'third'
```

if indices are negative numbers, they indicate positions counting from the end of the list, with –1 being the last position in the list, –2 being the second-to-last position, and so forth.

```
>>> a = x[-1]
>>> a
'fourth'
>>> x[-2]
'third'
```

List indices

- In thelist ["first", "second", "third", "fourth"], you can think of the indices as pointing like this:

x =[		"first",		"second",		"third",		"fourth"		]
Positive indices	0		1		2		3			
Negative indices	–4		–3		–2		–1			

Slicing in List

- enter list[index1:index2] to extract all items including index1 and up to (but not including) index2 into a new list.

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
```

Slicing in List

- When slicing a list, it's also possible to leave out index1 or index2. Leaving out index1 means "go from the beginning of the list," and leaving out index2 means "go to the end of the list":

```
>>> x[:3]
['first', 'second', 'third']
>>> x[2:]
['third', 'fourth']
```

Slicing in List

- Omitting both indices makes a new list that goes from the beginning to the end of the original list; that is, it copies the list. This is useful when you wish to make a copy that you can modify, without affecting the original list:

```
>>> y = x[:]
>>> y[0] = '1 st'
>>> y
['1 st', 'second', 'third', 'fourth']
>>> x
['first', 'second', 'third', 'fourth']
```

Modifying lists

- You can use list index notation to modify a list as well as to extract an element from it.

```
>>> x = [1, 2, 3, 4]
>>> x[1] = "two"
>>> x
[1, 'two', 3, 4]
```

Modifying lists

- Slice notation can be used here too. Saying something like lista[index1:index2] = listb causes all elements of lista between index1 and index2 to be replaced with the elements in listb. listb can have more or fewer elements than are removed from lista, in which case the length of lista will be altered. You can use slice assignment to do a number of different things, as shown here:

```
>>> x = [1, 2, 3, 4]
>>> x[len(x):] = [5, 6, 7]
>>> x
[1, 2, 3, 4, 5, 6, 7]
>>> x[:0] = [-1, 0]
>>> x
[-1, 0, 1, 2, 3, 4, 5, 6, 7]
>>> x[1:-1] = []
>>> x
[-1, 7]
```

Appends list to end of list

Appends list to front of list

Removes elements from list

Append, Extend

- Appending a single element to a list is such a common operation that there's a special append method to do it:

```
>>> x = [1, 2, 3]
>>> x.append("four")
>>> x
[1, 2, 3, 'four']
```

The extend method is like the append method, except that it allows you to add one list to another:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 5, 6, 7]
```

Output…

```
1   fruits = ['apples', 'oranges', 'bananas']
2   num = [2, 9]
3   fruits.extend(num)
4   print fruits
```

ython Shell

Commands execute without debug. Use arrow keys for history.

```
2.7.6 | 64-bit | (default, Jun  4 2014, 16:
Python Type "help", "copyright", "credits"
[evaluate untitled-1.py]
```

Output…

```
1  fruits = ['apples', 'oranges', 'bananas']
2  num = [2, 9]
3  fruits.append(num)
4  print fruits
```

Python Shell

Commands execute without debug. Use arrow k

```
2.7.6 | 64-bit | (defau
Python Type "help", "co
[evaluate untitled-1.py
```

```
1  fruits = ['apples', 'oranges', 'bananas']
2  num = [2, 9]
3  fruits.append(num)
4  print fruits
5  print fruits[3]
6  print fruits[3][1]
```

Python Shell

Commands execute without debug. Use arrow keys for history.

```
2.7.6 | 64-bit | (default, Jun  4 2014, 16:
Python Type "help", "copyright", "credits"
[evaluate untitled-1.py]
```

Insert

- insert is used as a method of lists and takes two additional arguments; the first is the index position in the list where the new element should be inserted, and the second is the new element itself:

```
>>> x = [1, 2, 3]
>>> x.insert(2, "hello")
>>> print(x)
[1, 2, 'hello', 3]
>>> x.insert(0, "start")
>>> print(x)
['start', 1, 2, 'hello', 3]
```

```
>>> x = [1, 2, 3]
>>> x.insert(-1, "hello")
>>> print(x)
[1, 2, 'hello', 3]
```

The del statement

- The del statement is the preferred method of deleting list items or slices. It doesn't do anything that can't be done with slice assignment, but it's usually easier to remember and easier to read:

```
>>> x = ['a', 2, 'c', 7, 9, 11]
>>> del x[1]
>>> x
['a', 'c', 7, 9, 11]
>>> del x[:2]
>>> x
[7, 9, 11]
```

removes

- remove looks for the first instance of a given value in a list and removes that value from the list:

```
>>> x = [1, 2, 3, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 5]
>>> x.remove(3)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

Sorting lists

- Lists can be sorted using the built-in Python sort method:

```
>>> x = [3, 8, 4, 0, 2, 1]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4, 8]
```

```
>>> x = ["Life", "Is", "Enchanting"]
>>> x.sort()
>>> x
['Enchanting', 'Is', 'Life']
```

List membership with the in operator

- It's easy to test if a value is in a list using the in operator, which returns a Boolean value. You can also use the converse, the not in operator:

```
>>> 3 in [1, 3, 4, 5]
True
>>> 3 not in [1, 3, 4, 5]
False
>>> 3 in ["one", "two", "three"]
False
>>> 3 not in ["one", "two", "three"]
True
```

*+ operator and * operator*

- To create a list by concatenating two existing lists, use the + (list concatenation) operator. This will leave the argument lists unchanged.

```
>>> z = [1, 2, 3] + [4, 5]
>>> z
[1, 2, 3, 4, 5]
```

```
>>> z = [None] * 4
>>> z
[None, None, None, None]
```

Min/Max

- You can use min and max to find the smallest and largest elements in a list

```
>>> min([3, 7, 0, -2, 11])
-2
>>> max([3, 7, 0, -2, 11])
11
```

List search with index

- If you wish to find where in a list a value can be found (rather than wanting to know only if the value is in the list), use the index method.

```
>>> x = [1, 3, "five", 7, -2]
>>> x.index(7)
3
>>> x.index(5)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

List matches with count

- count also searches through a list, looking for a given value, but it returns the number of times that value is found in the list rather than positional information:

```
>>> x = [1, 2, 2, 3, 5, 2, 5]
>>> x.count(2)
3
>>> x.count(5)
2
>>> x.count(4)
0
```

Summary of list operations

List operation	Explanation	Example
`[]`	Creates an empty list	`x = []`
`len`	Returns the length of a list	`len(x)`
`append`	Adds a single element to the end of a list	`x.append('y')`
`insert`	Inserts a new element at a given position in the list	`x.insert(0, 'y')`
`del`	Removes a list element or slice	`del(x[0])`
`remove`	Searches for and removes a given value from a list	`x.remove('y')`
`reverse`	Reverses a list in place	`x.reverse()`

Summary of list operations

List operation	Explanation	Example
`sort`	Sorts a list in place	`x.sort()`
`+`	Adds two lists together	`x1 + x2`
`*`	Replicates a list	`x = ['y'] * 3`
`min`	Returns the smallest element in a list	`min(x)`
`max`	Returns the largest element in a list	`max(x)`
`index`	Returns the position of a value in a list	`x.index['y']`
`count`	Counts the number of times a value occurs in a list	`x.count('y')`
`in`	Returns whether an item is in a list	`'y' in x`

Nested lists

- Lists can be nested. One application of this is to represent two-dimensional matrices. The members of these can be referred to using two-dimensional indices. Indices for these work as follows:

```
>>> m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
>>> m[0]
[0, 1, 2]
>>> m[0][1]
1
>>> m[2]
[20, 21, 22]
>>> m[2][2]
22
```

Thanks