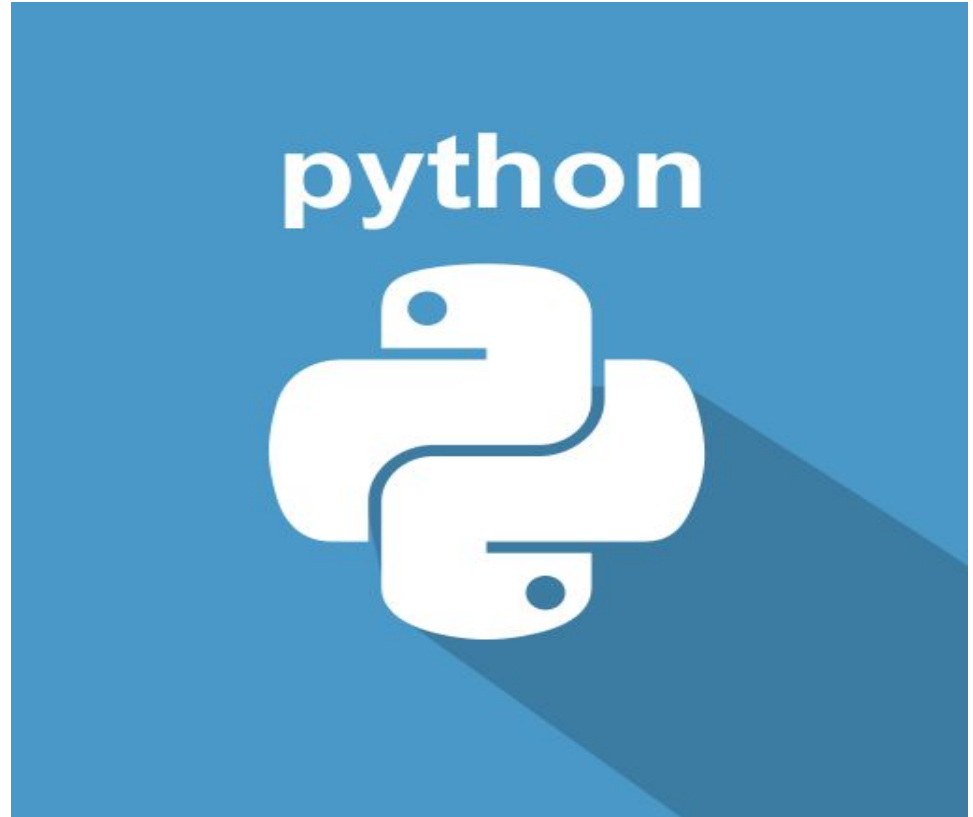


Lecture 3

Introduction to Computers and Python Programming

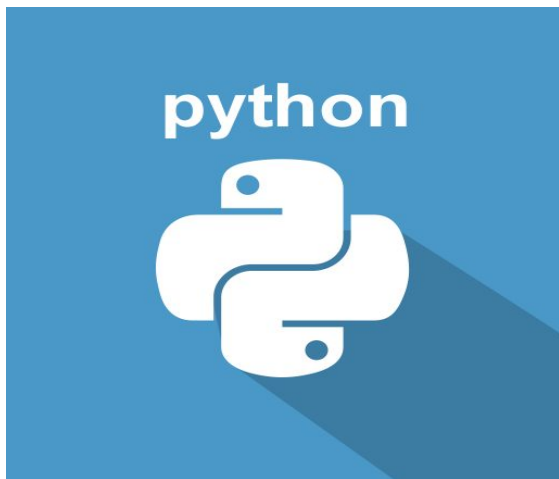


Last Time

- **Introduced Ourselves**
- **Discussed Hardware and Software**
- **Learned How Computers Store Data**
- **Discussed How Programs Work**
 - **Fetch-Decode-Execute Cycle**
- **Introduced Python**

Topics

- **Installing Python Stack with Anaconda**
- **Installing and Using Jupyter Notebooks**
- **Getting Started With Python**
- **First Python Examples -- Visualizing Data**
- **An Exercise -- Finding Nearest Neighbors**



matplotlib

matplotlib

- **Matplotlib is a python library that allows you to visualize data and mathematical functions**
- **It comes with tools for making plots and graphs**

matplotlib

- Installation would normally use one of the following commands

`pip install matplotlib`

`conda install matplotlib`

- You most likely installed the full version of anaconda in which case matplotlib is already installed

matplotlib

To use matplotlib we import the library as we learned earlier

```
import matplotlib.pyplot as plt  
%matplotlib inline
```

The second line is called a “magic” and is specific to jupyter notebooks. It tells the notebook to automatically display matplotlib plots.

matplotlib

Let's generate some data for us to plot

Generate points between -2 and 2 and store them in the variable x

x = np.linspace(-2,2, 200)

Evaluate the squares of each x value and store them in the variable y2

y2 = x2**

Evaluate the cubes of each x value and store them in the variable y3

y3 = x3**

matplotlib

- `plt.plot(...)` will plot our data
 - `c = '...'` determines the color of the plot
 - `label = '...'` determines the label (used in the legend)
- `plt.xlabel()`, `plt.ylabel()` labels the x and y axes
- `plt.title('...')` titles the plot
- `plt.legend()` shows the legend

matplotlib

plot y2 vs x and give it the color blue by using the c parameter

```
plt.plot(x, y2, c='blue', label=r'y =  $x^2$ ')
```

```
plt.plot(x, y3, c='red', label=r'y =  $x^3$ ')
```

label the x-axis

```
plt.xlabel('x')
```

label the y-axis

```
plt.ylabel('y')
```

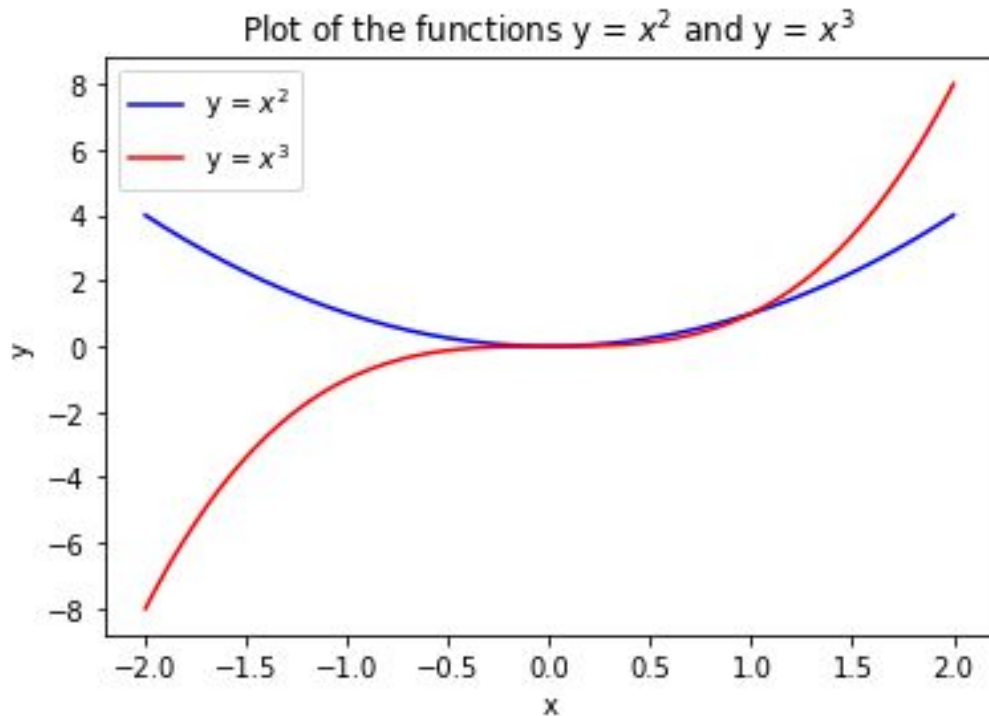
title the plot

```
plt.title(r'Plot of the functions y =  $x^2$  and y =  $x^3$ ')
```

show the legend

```
plt.legend()
```

matplotlib



matplotlib

```
plt.suptitle(r'Plot of the functions  $y = x^2$  and  $y = x^3$ ') # title the plot
```

```
plt.subplot(1, 2, 1) # create first subplot
```

```
plt.plot(x, y2, c='blue', label=r' $y = x^2$ ') # plot the first function
```

```
plt.xlabel('x') # label subplot x-axis
```

```
plt.ylabel('y') # label subplot y-axis
```

```
plt.legend() # show the subplot legend
```

```
plt.subplot(1, 2, 2) # create the second subplot
```

```
plt.plot(x, y3, c='red', label=r' $y = x^3$ ') #plot the second function
```

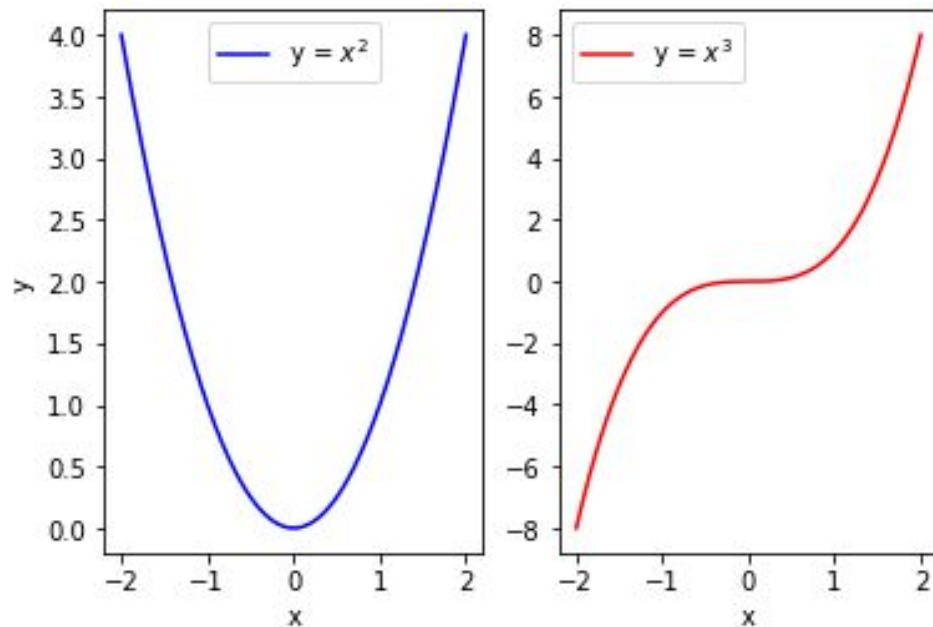
```
plt.xlabel('x') # label subplot x-axis
```

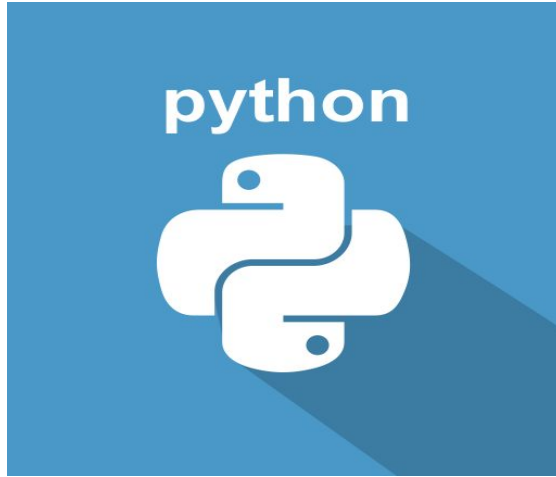
```
plt.ylabel('y') # label subplot y-axis
```

```
plt.legend() # show the subplot legend
```

matplotlib

Plot of the functions $y = x^2$ and $y = x^3$





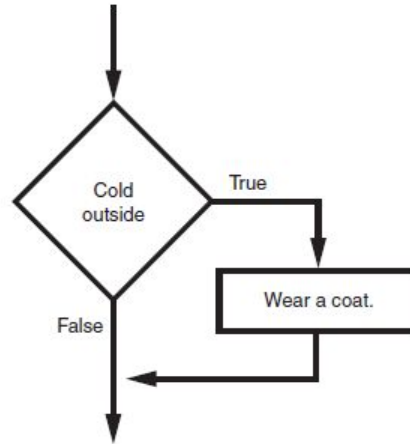
Decision Structures

The if Statement

- Control structure: logical design that controls order in which set of statements execute
- Sequence structure: set of statements that execute in the order they appear
- Decision structure: specific action(s) performed only if a condition exists

The if Statement

Figure 3-1 A simple decision structure



The if Statement

- **Python syntax:**

if condition:

Statement

Statement

The if Statement

- **First line known as the if clause**
 - Includes the keyword if followed by condition
 - The condition can be true or false
 - When the if statement executes, the condition is tested, and if it is true the block statements are executed. otherwise, block statements are skipped

Boolean Expressions and Relational Operators

- **Boolean expression**: expression tested by if statement to determine if it is true or false
 - Example: $a > b$
 - true if a is greater than b; false otherwise
- **Relational operator**: determines whether a specific relationship exists between two values
 - Example: greater than ($>$)

Boolean Expressions and Relational Operators

- **>= and <= operators test more than one relationship**
 - It is enough for one of the relationships to exist for the expression to be true
- **== operator determines whether the two operands are equal to one another**
 - Do not confuse with assignment operator (=)
- **!= operator determines whether the two operands are not equal**

Boolean Expressions and Relational Operators

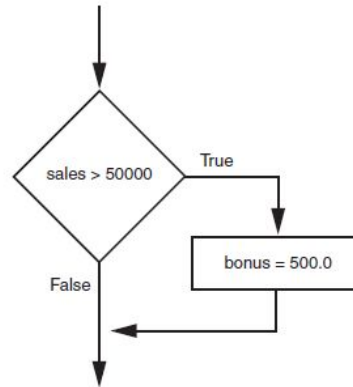
Table 3-2 Boolean expressions using relational operators

Expression	Meaning
<code>x > y</code>	Is x greater than y?
<code>x < y</code>	Is x less than y?
<code>x >= y</code>	Is x greater than or equal to y?
<code>x <= y</code>	Is x less than or equal to y?
<code>x == y</code>	Is x equal to y?
<code>x != y</code>	Is x not equal to y?

Boolean Expressions and Relational Operators

Using a Boolean expression with the $>$ relational operator

Figure 3-3 Example decision structure



Boolean Expressions and Relational Operators

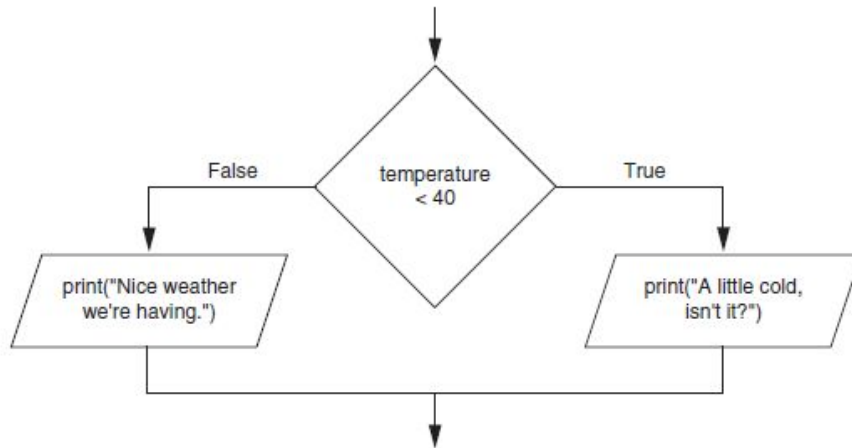
- **Any relational operator can be used in a decision block**
 - Example: `if balance == 0`
 - Example: `if payment != balance`
- **It is possible to have a block inside another block**
 - Example: `if` statement inside a function
 - Statements in inner block must be indented with respect to the outer block

The if-else Statement

- **Dual alternative decision structure**: two possible paths of execution
 - One is taken if the condition is true, and the other if the condition is false
 - Syntax: *if condition:*
 statements
 else:
 other statements
 - if clause and else clause must be aligned
 - Statements must be consistently indented

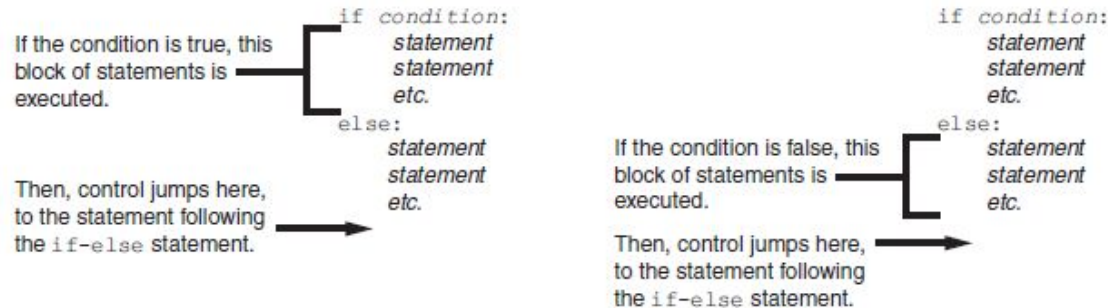
The if-else Statement

Figure 3-5 A dual alternative decision structure



The if-else Statement

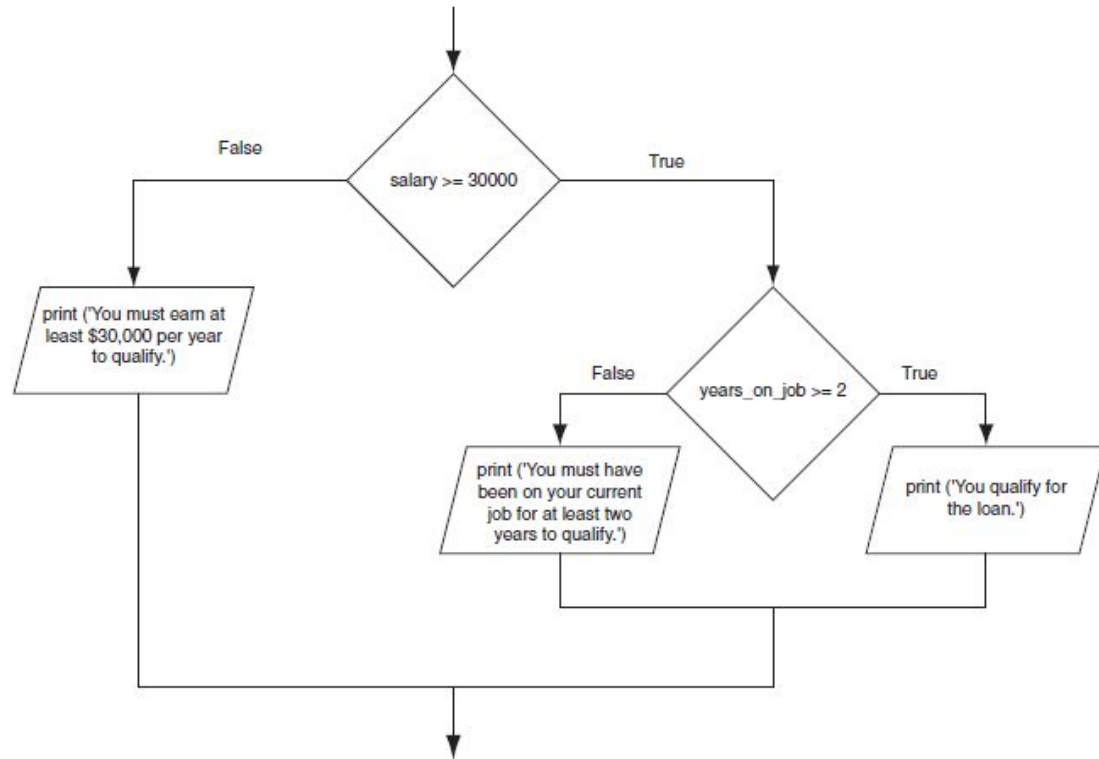
Figure 3-6 Conditional execution in an if-else statement



Nested Decision Structures

- **A decision structure can be nested inside another decision structure**
 - Commonly needed in programs
 - Example:
 - Determine if someone qualifies for a loan, they must meet two conditions:
 - Must earn at least \$30,000/year
 - Must have been employed for at least two years
 - Check first condition, and if it is true, check second condition

Figure 3-12 A nested decision structure



Nested Decision Structures

- **Important to use proper indentation in a nested decision structure**
 - Important for Python interpreter
 - Makes code more readable for programmer
 - Rules for writing nested if statements:
 - else clause should align with matching if clause
 - Statements in each block must be consistently indented

The if-elif-else Statement

- if-elif-else statement: special version of a decision structure
 - Makes logic of nested decision structures simpler to write
 - Can include multiple elif statements
 - Syntax: if condition1:
 statements
 elif condition2:
 statements
 else:
 statements

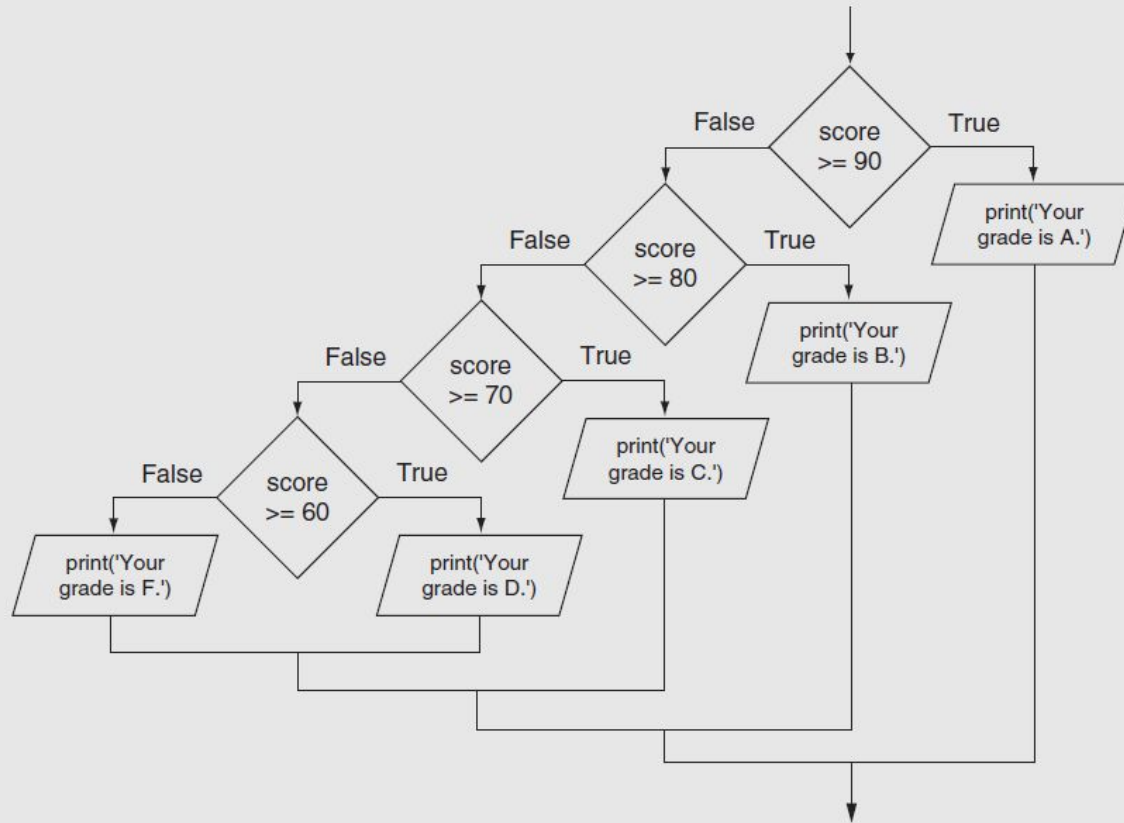
The if-elif-else Statement

- **Alignment used with if-elif-else statement:**
 - if, elif, and else clauses are all aligned
 - Conditionally executed blocks are consistently indented

The if-elif-else Statement

- **if-elif-else statement is never required, but logic easier to follow**
 - Can be accomplished by nested if-else
 - Code can become complex, and indentation can cause problematic long lines

Figure 3-15 Nested decision structure to determine a grade



Logical Operators

- **Logical operators**: operators that can be used to create complex Boolean expressions
 - and operator and or operator: binary operators, connect two Boolean expressions into a compound Boolean expression
 - not operator: unary operator, reverses the truth of its Boolean operand

The and Operator

- Takes two Boolean expressions as operands
 - Creates compound Boolean expression that is true only when both sub expressions are true
 - Can be used to simplify nested decision structures

The and Operator

Truth table for the and operator

Expression	Value of the Expression
false and false	false
false and true	false
true and false	false
true and true	true

The or Operator

- Takes two Boolean expressions as operands
 - Creates compound Boolean expression that is true when either of the sub expressions is true
 - Can be used to simplify nested decision structures

The or Operator

Truth table for the or operator

Expression	Value of the Expression
false or false	false
false or true	true
true or false	true
true or true	true

Short-Circuit Evaluation

- Short circuit evaluation: deciding the value of a compound Boolean expression after evaluating only one sub expression

Short-Circuit Evaluation

- **Performed by the or and and operators**
 - For or operator: If left operand is true, compound expression is true. Otherwise, evaluate right operand
 - For and operator: If left operand is false, compound expression is false. Otherwise, evaluate right operand

The not Operator

- **Takes one Boolean expressions as operand and reverses its logical value**
 - Sometimes it may be necessary to place parentheses around an expression to clarify to what you are applying the not operator

Boolean Variables

- **Boolean variable**: references one of two values, True or False
 - Represented by bool data type

Boolean Variables

- **Commonly used as flags**
 - Flag: variable that signals when some condition exists in a program
 - Flag set to False ☐ condition does not exist
 - Flag set to True ☐ condition exists

Comparing Strings

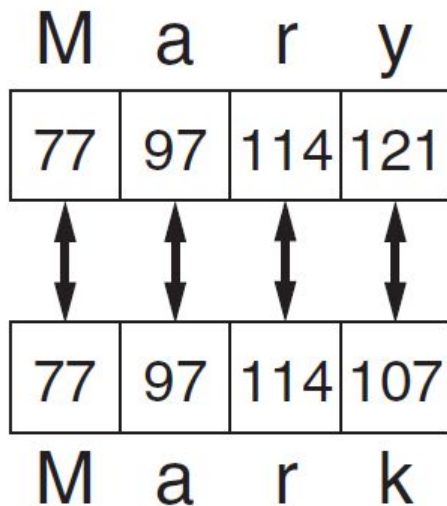
- **Strings can be compared using the == and != operators**
- **String comparisons are case sensitive**

Comparing Strings

- **Strings can be compared using $>$, $<$, $>=$, and $<=$**
 - Compared character by character based on the ASCII values for each character
 - If shorter word is substring of longer word, longer word is greater than shorter word

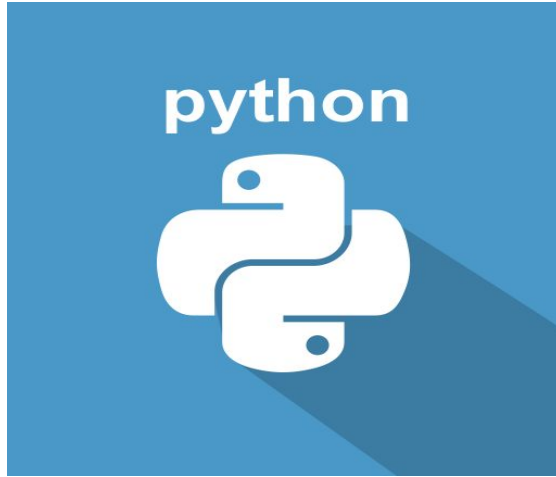
Comparing Strings

Figure 3-9 Comparing each character in a string



Summary

- Decision structures, including:
 - Single alternative decision structures
 - Dual alternative decision structures
 - Nested decision structures
- Relational operators and logical operators as used in creating Boolean expressions
- Boolean variables



Loops

Repetition Structures

- Often have to write code that performs the same task multiple times
 - Disadvantages to duplicating code
 - Makes program large
 - Time consuming
 - May need to be corrected in many places

Repetition Structures

- **Repetition structure**: makes computer repeat included code as necessary
 - Includes condition-controlled loops and count-controlled loops

The while Loop

while loop: while condition is true, do something

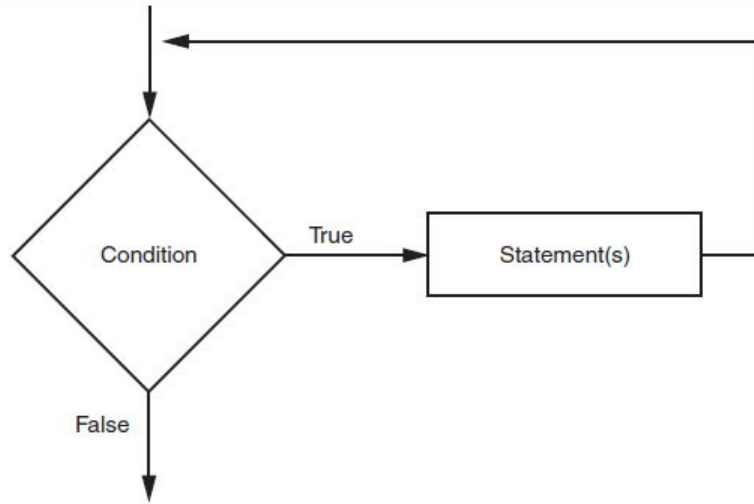
while condition:

Statements

- Condition tested for true or false value
- Statements repeated as long as condition is true

The while Loop

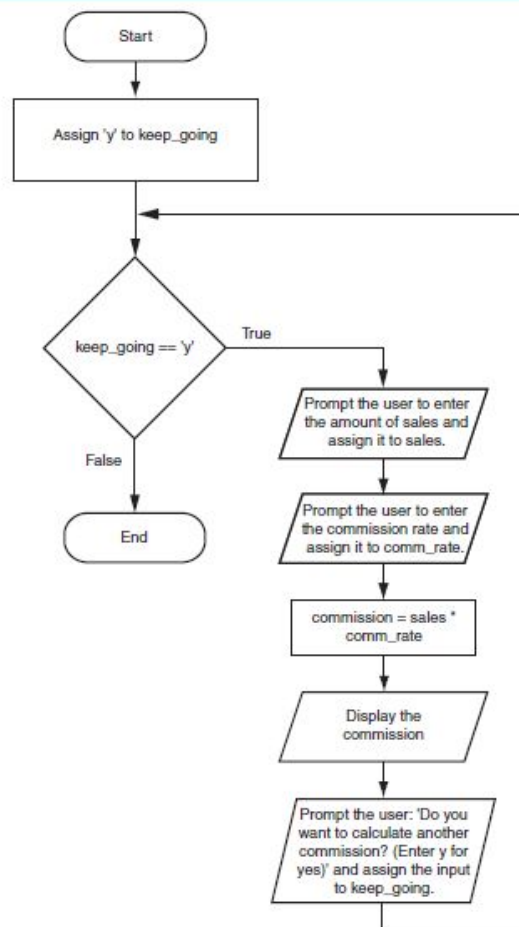
Figure 4-1 The logic of a `while` loop



The while Loop

- In order for a loop to stop executing, something has to happen inside the loop to make the condition false
- Iteration: one execution of the body of a loop

Figure 4-3 Flowchart for Program 4-1



Infinite Loops

- **Infinite loop**: loop that does not have a way of stopping
 - Repeats until program is interrupted
 - Occurs when programmer forgets to include stopping code in the loop

The for Loop

- Count-Controlled loop: iterates a specific number of times
- Use a for statement to write count-controlled loop


The for Loop


- Designed to work with sequence of data items
- Iterates once for each item in the sequence
- Target variable: the variable which is the target of the assignment at the beginning of each iteration


for variable in [val1, val2, etc]:


Statements


Figure 4-4 The for loop

1st iteration:  `for num in [1, 2, 3, 4, 5]:`
`print(num)`

2nd iteration:  `for num in [1, 2, 3, 4, 5]:`
`print(num)`

3rd iteration:  `for num in [1, 2, 3, 4, 5]:`
`print(num)`

4th iteration:  `for num in [1, 2, 3, 4, 5]:`
`print(num)`

5th iteration:  `for num in [1, 2, 3, 4, 5]:`
`print(num)`

Using range

- **The range function simplifies the process of writing a for loop**
 - range returns an iterable object
 - Iterable: contains a sequence of values that can be iterated over

Using range

- **range arguments:**
 - One argument:
 - Starts from 0, argument used as ending limit
 - Two arguments:
 - starting value and ending limit
 - Three arguments:
 - third argument is step value

Using enumerate

- Allows keeping track of data and index in for loop

```
for idx, value in enumerate(data):
```

```
    ...
```

Using range

- **The range function can be used to generate a sequence with numbers in descending order**
 - Make sure starting number is larger than end limit, and step value is negative
 - Example: `range (10, 0, -1)`

Augmented Assignment

- In many assignment statements, the variable on the left side of the = operator also appears on the right side of the = operator

Augmented Assignment

- Augmented assignment operators:
special set of operators designed for this type of job
 - Shorthand operators

Augmented Assignment

Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>-=</code>	<code>y -= 2</code>	<code>y = y - 2</code>
<code>*=</code>	<code>z *= 10</code>	<code>z = z * 10</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>%=</code>	<code>c %= 3</code>	<code>c = c % 3</code>

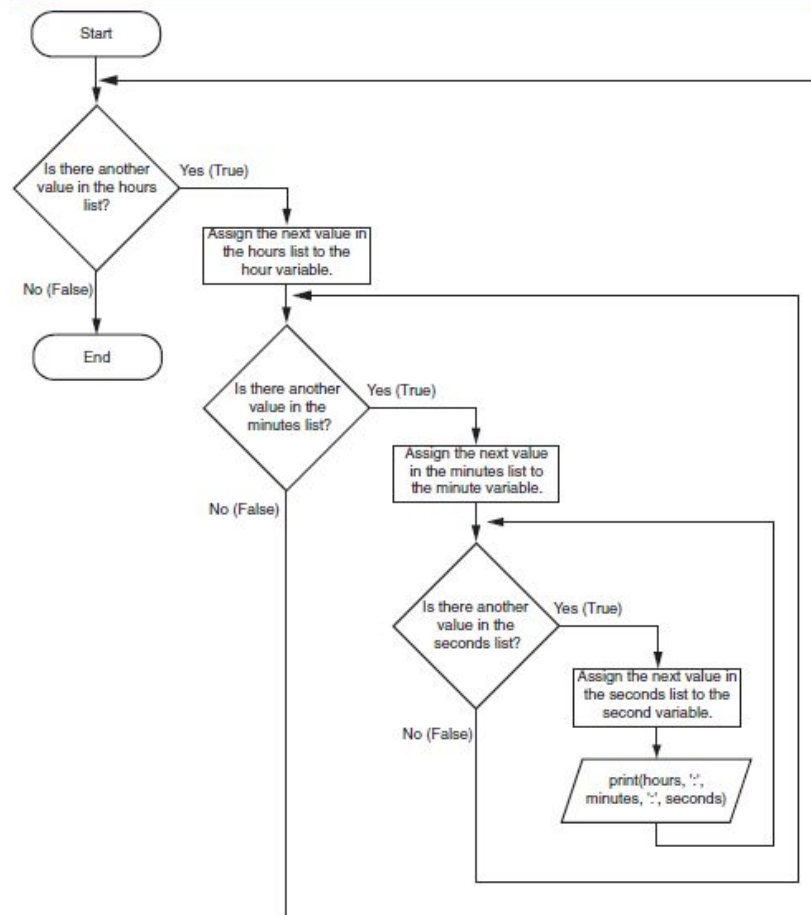
Nested Loops

- **Nested loop**: loop that is contained inside another loop

Nested Loops

- Example: analog clock works like a nested loop
 - Hours hand moves once for every twelve movements of the minutes hand: for each iteration of the “hours,” do twelve iterations of “minutes”
 - Seconds hand moves 60 times for each movement of the minutes hand: for each iteration of “minutes,” do 60 iterations of “seconds”

Figure 4-8 Flowchart for a clock simulator

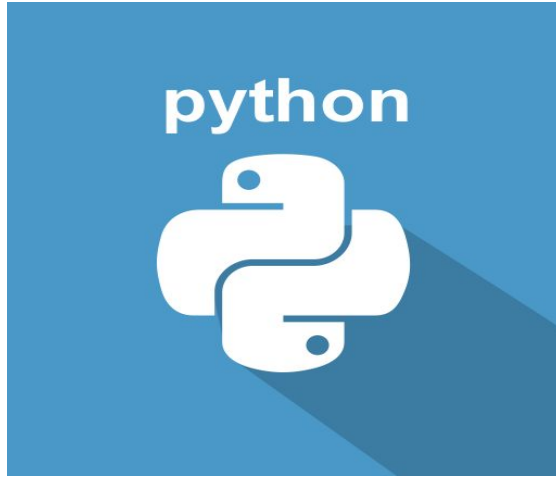


Nested Loops

- **Key points about nested loops:**
 - Inner loop goes through all of its iterations for each iteration of outer loop
 - Inner loops complete their iterations faster than outer loops
 - Total number of iterations in nested loop:
 $\text{number_iterations_inner} \times \text{number_iterations_outer}$

Summary

- Repetition structures, including:
 - Condition-controlled loops
 - Count-controlled loops
 - Nested loops
- Infinite loops and how they can be avoided
- range function as used in for loops



Functions

Introduction to Functions

- **Function**: group of statements within a program that perform as specific task
 - Usually one task of a large program
 - Functions can be executed in order to perform overall program task
 - Known as *divide and conquer* approach

[illegible]

```
def function1():
    statement
    statement
    statement
```

function

```
def function2():
    statement
    statement
    statement
```

function

```
def function3():
    statement
    statement
    statement
```

function

```
def function4():
    statement
    statement
    statement
```

function

Benefits of Modularizing

- **The benefits of using functions include:**
 - Simpler code
 - Code reuse
 - Better testing and debugging
 - Faster development

Void Function

- A void function:
 - Simply executes the statements it contains and then terminates.
 - No return value (technically returns *None*)

Value-Returning Functions

- A value-returning function:
 - Executes the statements it contains, and then it returns a value back to the statement that called it.
 - Return value can be any python type

Function names

- Function naming rules:
 - Cannot use key words as a function name
 - Cannot contain spaces
 - First character must be a letter or underscore
 - All other characters must be a letter, number or underscore
 - Uppercase and lowercase characters are distinct

Defining a Function

- **Function definition:** specifies what function does

```
def function_name():  
    statement  
    statement
```

Calling a Function

- **Call a function to execute it**
 - When a function is called:
 - Interpreter jumps to the function and executes statements in the block
 - Interpreter jumps back to part of program that called the function
 - Known as function return

Main

- main function: called when the program starts
 - Calls other functions when they are needed
 - Defines the *mainline logic* of the program
 - Can be implicit in scripts

Indentation in Python

- **Each block must be indented**
 - Lines in block must begin with the same number of spaces
 - Use tabs or spaces to indent lines in a block, but not both as this can confuse the Python interpreter
 - Blank lines that appear in a block are ignored

Global Variables

- **Global variable**: created by assignment statement written outside all the functions
 - Can be accessed by any statement in the program file, including from within a function

Global Variables

If a function needs to assign a value to the global variable, the global variable must be redeclared within the function

`global variable_name`

Global Variables

- **Reasons to avoid using global variables:**
 - Global variables making debugging difficult
 - Functions that use global variables are difficult to reuse
 - Global variables make a program hard to understand

Local Variables

- **Local variable**: variable that is assigned a value inside a function
 - Belongs to the function in which it was created
 - Only statements inside that function can access it, error will occur if another function tries to access the variable

Local Variables

- **Scope**: the part of a program in which a variable may be accessed
 - For local variable: function in which created

Local Variables


- **Local variable cannot be accessed by statements outside of its scope**
- **As a result different functions may have local variables with the same name**
 - Each function does not see the other function's local variables

Passing Arguments to Functions

- **Argument**: piece of data that is sent into a function
 - Function can use argument in calculations
 - When calling the function, the argument is placed in parentheses following the function name

Passing Arguments to Functions

Figure 5-13 The value variable is passed as an argument

```
def main():  
    value = 5  
    show_double(value)  
      
  
def show_double(number):  
    result = number * 2  
    print(result)
```

Passing Arguments to Functions

- Parameter variable: variable that is assigned the value of an argument when the function is called

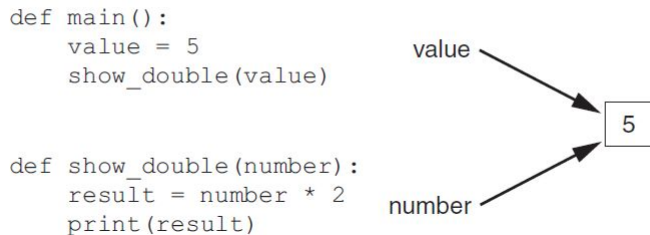
```
def function_name(parameter):
```

```
...
```

- Scope of a parameter is the function in which the parameter is used

Passing Arguments to Functions

Figure 5-14 The `value` variable and the `number` parameter reference the same value



Passing Multiple Arguments

- **Python allows writing a function that accepts multiple arguments**
 - Parameter list replaces single parameter

Passing Multiple Arguments

- Arguments can be passed *by position* to corresponding parameters
 - First parameter receives value of first argument, second parameter receives value of second argument, etc.

Passing Multiple Arguments

Figure 5-16 Two arguments passed to two parameters

```
def main():  
    print('The sum of 12 and 45 is')  
    show_sum(12, 45)
```

```
def show_sum(num1, num2):  
    result = num1 + num2  
    print(result)
```



Function Parameters

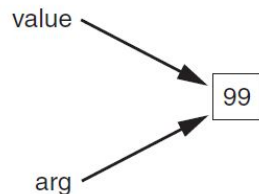
- Changes made to a parameter value within the function do not affect the argument
 - Known as *pass by value*
 - Provides a way for unidirectional communication between one function and another function

Making Changes to Parameters

Figure 5-17 The value variable is passed to the change_me function

```
def main():  
    value = 99  
    print('The value is', value)  
    change_me(value)  
    print('Back in main the value is', value)
```

```
def change_me(arg):  
    print('I am changing the value.')  
    arg = 0  
    print('Now the value is', arg)
```

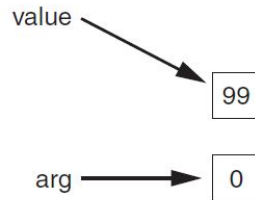


Function Parameters

Figure 5-18 The value variable is passed to the change_me function

```
def main():  
    value = 99  
    print('The value is', value)  
    change_me(value)  
    print('Back in main the value is', value)
```

```
def change_me(arg):  
    print('I am changing the value.')  
    arg = 0  
    print('Now the value is', arg)
```



Keyword Arguments

- **Keyword argument:** argument that specifies which parameter the value should be passed to
 - Position when calling function is irrelevant
 - General Format:

`function_name(parameter=value)`

Keyword Arguments

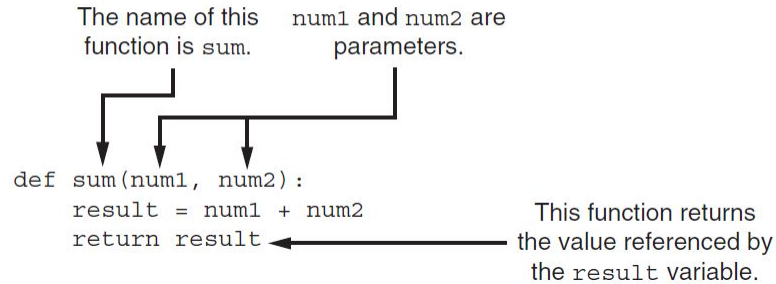
- **Possible to mix keyword and positional arguments when calling a function**
 - Positional arguments must appear first

Return values

- To write a value-returning function, you write a simple function and add one or more return statements
 - Format: return *expression*
 - The value for *expression* will be returned to the part of the program that called the function

Return values

Figure 5-23 Parts of the function



Returning Strings

- You can write functions that return strings

```
def get_name():  
    # Get the user's name.  
    name = input('Enter your name: ')  
    # Return the name.  
    return name
```

Returning Boolean Values

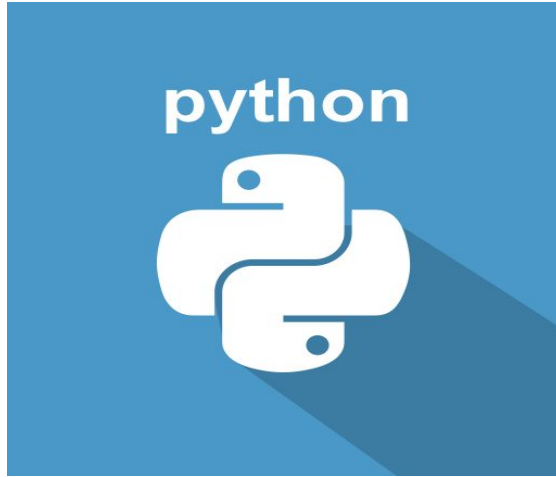
- **Boolean function**: returns either True or False
 - Use to test a condition such as for decision and repetition structures
 - Common calculations, such as whether a number is even, can be easily repeated by calling a function
 - Use to simplify complex input validation code

Returning Multiple Values

- In Python, a function can return multiple values
 - Specified (as a tuple) after the return statement separated by commas
 - Format: return *expression1*, *expression2*, etc.

Summary

- The advantages of using functions
- The syntax for defining and calling a function
- Use of local variables and their scope
- Syntax and limitations of passing arguments to functions



Sequences

Sequences

- **Sequence**: an object that contains multiple items of data
 - The items are stored in sequence one after another

Sequences

- **Python provides different types of sequences, including lists and tuples**
 - The difference between these is that a list is mutable and a tuple is immutable

Introduction to Lists

- **List**: an object that contains multiple data items
 - **Element**: An item in a list
 - **Format**: *list* = [*item1*, *item2*, etc.]
 - Can hold items of different types
- **list()** constructor converts certain types of objects to lists

Introduction to Lists

Figure 7-1 A list of integers

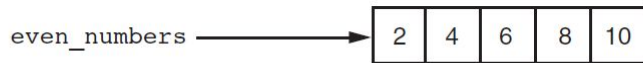


Figure 7-2 A list of strings



Figure 7-3 A list holding different types



The Repetition Operator

- **Repetition operator**: makes multiple copies of a list and joins them together
 - The * symbol is a repetition operator when applied to a sequence and an integer

*list * n*

Iterating over a List

- **We've seen that you can iterate over a list using a `for` loop**

```
for x in list:
```

```
    . . .
```


Indexing

- **Index**: a number specifying the position of an element in a list
 - Enables access to individual element in list

Indexing

- Index of first element in the list is 0, second element is 1, and n 'th element is $n-1$
- Negative indexes identify positions relative to the end of the list
 - The index -1 identifies the last element, -2 identifies the next to last element, etc.

The len function

- len function: returns the length of a sequence such as a list
 - Example: `size = len(my_list)`
 - Returns the number of elements in the list, so the index of last element is `len(list)-1`

Lists Are Mutable

- **Mutable sequence: the items in the sequence can be changed**
 - Lists are mutable, and so their elements can be changed

Lists Are Mutable

- An expression such as
`list[1] = new_value` can be used to
assign a new value to a list element

Concatenating Lists

- Concatenate: join two things together
- The + operator can be used to concatenate two lists
 - Be careful when trying to concatenate a list with a non-sequence

List Slicing

- **Slice**: a span of items that are taken from a sequence

list[start : end : step]

List Slicing

Slicing returns a list containing copies of elements from *start* up to, but not including, *end*

- If *start* not specified, 0 is used for start index
- If *end* not specified, `len(list)` is used for end index

Slicing expressions can include a step value and negative indexes relative to end of list

The in Operator

- You can use the in operator to determine whether an item is contained in a list
 - General format: *item in list*
 - Returns True if the item is in the list, or False if it is not in the list

List Methods

- **index(*item*)**: used to determine where an item is located in a list
 - Returns the index of the first element in the list containing item

List Methods

- **append(*item*)**: used to add items to a list
 - *item* is appended to the end of the existing list

List Methods

- **insert(*index*, *item*)**: used to insert *item* at position *index* in the list
- **sort()**: used to sort the elements of the list in ascending order

List Methods

- **remove(*item*)**: removes the first occurrence of *item* in the list
- **reverse()**: reverses the order of the elements in the list

Table 7-1 A few of the list methods

Method	Description
<code>append(<i>item</i>)</code>	Adds <i>item</i> to the end of the list.
<code>index(<i>item</i>)</code>	Returns the index of the first element whose value is equal to <i>item</i> . A <code>ValueError</code> exception is raised if <i>item</i> is not found in the list.
<code>insert(<i>index</i>, <i>item</i>)</code>	Inserts <i>item</i> into the list at the specified <i>index</i> . When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
<code>sort()</code>	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
<code>remove(<i>item</i>)</code>	Removes the first occurrence of <i>item</i> from the list. A <code>ValueError</code> exception is raised if <i>item</i> is not found in the list.
<code>reverse()</code>	Reverses the order of the items in the list.

Useful Built-in Functions

- **del**: removes an element from a specific index in a list

`del list[i]`

- **min and max**: built-in functions that return the lowest or highest value in a sequence
 - The sequence is passed as an argument

Two-Dimensional Lists

- **Two-dimensional list: a list that contains other lists as its elements**
 - Also known as nested list
 - Common to think of two-dimensional lists as having rows and columns
- **To process data in a two-dimensional list need to use two indexes**

Two-Dimensional Lists

Figure 7-5 A two-dimensional list

	Column 0	Column 1
Row 0	'Joe'	'Kim'
Row 1	'Sam'	'Sue'
Row 2	'Kelly'	'Chris'

Two-Dimensional Lists

Figure 7-7 Subscripts for each element of the `scores` list

	Column 0	Column 1	Column 2
Row 0	<code>scores[0][0]</code>	<code>scores[0][1]</code>	<code>scores[0][2]</code>
Row 1	<code>scores[1][0]</code>	<code>scores[1][1]</code>	<code>scores[1][2]</code>
Row 2	<code>scores[2][0]</code>	<code>scores[2][1]</code>	<code>scores[2][2]</code>

Tuples

- Tuples are an immutable sequence
 - Very similar to a list
 - Once it is created it cannot be changed

(item 1, item 2, ..., item n)

Tuples

- Tuples support operations as lists
 - Subscript indexing for retrieving elements
 - Methods such as index
 - Built in functions such as len, min, max
 - Slicing expressions
 - The in, +, and * operators

Tuples

- **Tuples do not support methods inducing mutation:**
 - append
 - remove
 - insert
 - reverse
 - sort

Tuples

- **Advantages for using tuples over lists:**
 - Processing tuples is faster than processing lists
 - Tuples are safe (because of immutability)
 - Some operations (typically ones requiring hashing or immutability) in Python need tuples

Tuples

list() constructor: converts sequence to list

tuple() constructor: converts sequence to tuple

Arrays

- **Arrays are a mutable memory contiguous sequence of elements of the same type**
 - Similar to a list (but can't store items different types)

Arrays

```
import array
```

```
array.array('i',[1, 2, 3, 1, 5])
```

Tuples

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Summary

- Lists, including:
 - Repetition and concatenation operators
 - Indexing
 - Slicing
 - List methods and built-in functions
 - Two-dimensional lists
- Tuples, including:
 - Immutability
 - Difference with/advantages over lists