پوهنتون کاردان
KARDAN UNIVERSITY

MODERN PROGRAMMING LANGUAGE (Python)

Starting Programming in Python

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 (cont'd.)

- **>= 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 (cont'd.)

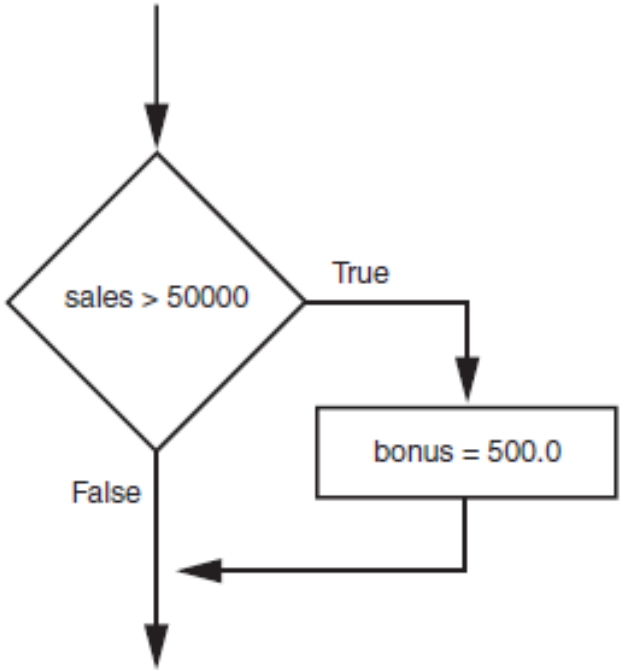
Table 4-2 Boolean expressions using relational operators

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

Boolean Expressions and Relational Operators (cont'd.)

- **Using a Boolean expression with the > relational operator**

Figure 4-3 Example decision structure



Boolean Expressions and Relational Operators (cont'd.)

- **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

Bitwise operators

OPERATOR	NAME	DESCRIPTION	SYNTAX
&	Bitwise AND	Result bit 1,if both operand bits are 1;otherwise results bit 0.	x & y
|	Bitwise OR	Result bit 1,if any of the operand bit is 1; otherwise results bit 0.	x | y
~	Bitwise NOT	inverts individual bits	~x
^	Bitwise XOR	Results bit 1,if any of the operand bit is 1 but not both, otherwise results bit 0.	x ^ y
>>	Bitwise right shift	The left operand's value is moved toward right by the number of bits specified by the right operand.	x>>
<<	Bitwise left shift	The left operand's value is moved toward left by the number of bits specified by the right operand.	x<<

Identity Operators

- Identity operators compare the memory locations of two objects

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Membership operators

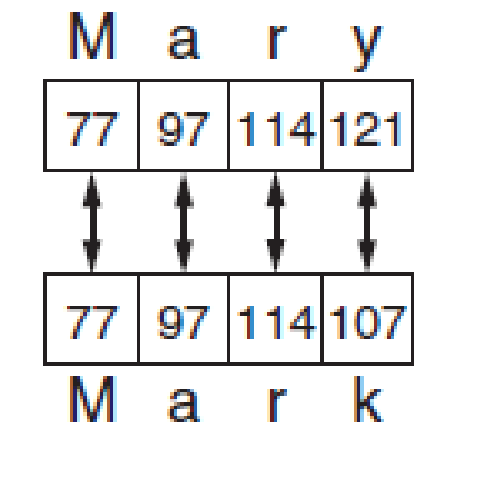
- Python offers two membership operators to check or validate the membership of a value.
- It tests for membership in a sequence, such as strings, lists, or tuples
- **in operator:** The ‘in’ operator is used to check if a character/ substring/ element exists in a sequence or not
- **‘not in’ operator-** Evaluates to true if it does not finds a variable in the specified sequence and false otherwise

Comparing Strings

- **Strings can be compared using the == and != operators**
- **String comparisons are case sensitive**
- **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 (cont'd.)

Figure 4-11 Comparing each character in a string



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
- **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
- **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**
  - 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
- **Truth table for the `not` operator**

Expression	Value of the Expression
true	false
false	true

Checking Numeric Ranges with Logical Operators

- **To determine whether a numeric value is within a specific range of values, use `and`**
  - Example: `x >= 10 and x <= 20`
- **To determine whether a numeric value is outside of a specific range of values, use `or`**
  - Example: `x < 10 or x > 20`

Boolean Variables

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

پوهنتون کاردان
KARDAN UNIVERSITY

MODERN PROGRAMMING LANGUAGE
(Python)

Starting Programming in Python
(Data Types/Structures)

Learning Outcome

- Should know about The python Data Types and Objects
- Should know about python Numbers
- Should learn about python Strings
- Should learn Lists in Python
- Should Learn Dictionaries in Python
- Should Know about python Tuples
- Should Know about Python Sets
- I/O with Python

The different built-in types

Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

Numbers

- Two types:
  - Integers
  - Floating Numbers

```
my_income = 100

tax_rate = 0.1

my_taxes = my_income * tax_rate
```

Numbers

```
2 ** 100                    # 2 to the power 100, again
```

- For a floor division use double //
- 18//4
  - output: 4
- For reminder use %
- 18%4
  - Output: 2

Strings

- Strings are the ordered sequence of characters.

```
In [3]: print ('This is the program \t in Python \n Welcome!')
```

```
This is the program      in Python
 Welcome!
```

```
name='Saeed'
```

```
name +' Comming  from Home'
```

```
'Saeed Comming  from Home'
```

- You can concatenate the strings too.
- You can use either single (‘) or double quotes (“)

String

- To know about length of a String
  - len(String_name)
- To grab any index of a string
  - String_name[0] to grab first index
  - Ex: “Ahmad”, index 0 =A
  - A = “This is a String” A[-1]
  - # this is a comment in Python
- We can do String concatenation using plus (+) sign
  - Ex: str a=“Ahmad ”, str b=“ Naeem”
  - a+b= Ahmad Naeem
- String Multiplication
  - a=‘b’, a*4
  - a a a a

Contd..

- Finding a character

```
First='Samim'
First.find('a')
```

```
1
```

- Splitting the text

```
S='This is the first program in python'
```

```
S.split()
```

```
['This', 'is', 'the', 'first', 'program', 'in', 'python']
```

Print Statements

- Printing a string value in the console

```
a="Samim"

print(a+" is here!")
```

```
Samim is here!
```

- To print the values within the text

```
a='Ahmad'
b='Naeemi'
print("The name is {} and the Surname is {}".format(a,b))
```

```
The name is Ahmad and the Surname is Naeemi
```

- To print the values formatted

```
a='Ahmad'
b='Naeemi'
print(f"The name is {a} and the Surname is {b}")
```

```
The name is Ahmad and the Surname is Naeemi
```

Lists

- The Python list object is the most general *sequence*
- Lists are positionally ordered collections of arbitrarily typed objects

```
my_list=['Ahmad','Saeed','Samim',672,98.87 ]
```

```
my_list
```

```
['Ahmad', 'Saeed', 'Samim', 672, 98.87]
```

- Adding more values

```
my_list +[78,'Naeem']
```

```
['Ahmad', 'Saeed', 'Samim', 672, 98.87, 78, 'Naeem']
```

```
my_list[2]
```

```
'Samim'
```

Contd..

- Appending the values

```
my_list.append('Sadiq')
```

```
my_list
```

```
['Ahmad', 'Saeed', 'Samim', 672, 98.87, 'Sadiq']
```

- Deleting the values
  - Using pop

```
my_list.pop(3)
```

```
672
```

  - Using del

```
del my_list[2]
```

```
my_list
```

```
['Ahmad', 'Saeed', 672, 98.87, 'Sadiq']
```

List

- Inserting values in the list:

```
my_list.insert(3,' Herat')
```

```
my_list
```

```
['Ahmad', 'Saeed', 'Samim', ' Herat', 'Kabul', 98.87, 'Sadiq']
```

- Sorting the values
- (cannot sort between different types)

```
city_names=['Kabul','Herat','Balkh','Kandahar']
```

```
city_names
```

```
['Kabul', 'Herat', 'Balkh', 'Kandahar']
```

```
city_names.sort()
```

```
city_names
```

```
['Balkh', 'Herat', 'Kabul', 'Kandahar']
```

Dictionaries

- Dictionaries are different than sequences
- They are called Mappings
- They store objects by *key* instead of by relative position normally used to refer to the attributes and object of values stored in databases:
- The Key and Value is separated using “:”
- {‘Key’ : ’value’}

```
price_list={'Tomato': 30, 'Apple': 250, 'Milk': 50}
```

```
price_list
```

```
{'Tomato': 30, 'Apple': 250, 'Milk': 50}
```

- Here, the Tomato, Apple and Milk are keys.

Contd..

- Dictionaries cannot be sorted,
- You don’t have to know about index location

```
price_list
```

```
{'Tomato': 30, 'Apple': 250, 'Milk': 50}
```

```
price_list['Milk']
```

```
50
```

- We can add more keys and values to existent Dictionary:

```
price_list['Orange']=400
price_list
```

```
{'Tomato': 30, 'Apple': 250, 'Milk': 50, 'Orange': 400}
```

- We can find the collection of Keys, and Values
- price_list.Keys()…….. Price_list.values()

Tuples

- Tuples are *sequences*, like lists, but they are *immutable*
- They’re used to represent fixed collections of items

```
my_tuple=(5,2,3,43,232,'Sabawoon')
```

```
my_tuple
```

```
(5, 2, 3, 43, 232, 'Sabawoon')
```

- The other rules of a list is implemented on tuple
- Two specific methods:
  - my_tuple.count(value)
  - My_tuple.index(value)

Sets

- Unordered collections of unique elements

```
my_set=set()

my_set.add(1)
my_set
my_set.add("yes")
my_set
```

```
{1, 'yes'}
```

- We can add repeated values on lists but not on sets

```
my_lists=[1,2,2,2,1,1,2]
my_lists
```

```
[1, 2, 2, 2, 1, 1, 2]
```

```
my_set=set(my_lists)
my_set
```

```
{1, 2}
```

IO with Files

- File objects are Python code's main interface to external files on your computer.

Writing into a file

```
file=open('new.txt','w')
```

```
file.write("this is the first line ")
```

```
23
```

```
file.close()
```

Reading from a file

```
read=open('new.txt')
```

```
read.read()
```

```
'this is the first line '
```

- file.seek(0)…… to return the curser back to the first line

Files

- To get access to the files of different locations:

- Find the location of current file:
  - pwd

- File=open('C:\\Users\\Desktop\\New Folder\\read.txt', 'w')

More file operations

Reading, Writing, Appending Modes

- **mode='r'** is read only
- **mode='w'** is write only (will overwrite files or create new!)
- **mode='a'** is append only (will add on to files)
- **mode='r+'** is reading and writing
- **mode= 'w+'** is writing and reading (Overwrites existing files or creates a new file!)

A list of the different modes of opening a file:

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

A list of the different modes of opening a file:

wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

File Positions:

- The *tell()* method tells you the current position within the file in other words, the next read or write will occur at that many bytes from the beginning of the file:
- The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.
- If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example:

```
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
position = fo.tell();
print "Current file position : ", position
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
fo.close()
```

- This would produce following result:

```
Read String is : Python is
Current file position : 10
Again read String is : Python is
```

Renaming and Deleting Files:

- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.
- To use this module you need to import it first and then you can all any related functions.

The rename() Method:

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax:

```
os.rename(current_file_name, new_file_name)
```

Example:

```
import os
os.rename( "test1.txt", "test2.txt" )
```

The *delete()* Method:

You can use the *delete()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax:

```
os.remove(file_name)
```

Example:

```
import os
os.remove("test2.txt")
```

Directories in Python:

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The *mkdir()* Method:

You can use the *mkdir()* method of the os module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Syntax:

```
os.mkdir("newdir")
```

Example:

```
import os # Create a directory "test"
os.mkdir("test")
```

The *chdir()* Method:

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax:

```
os.chdir("newdir")
```

Example:

```
import os
os.chdir("/home/newdir")
```

The *getcwd()* Method:

The *getcwd()* method displays the current working directory.

Syntax:

```
os.getcwd()
```

Example:

```
import os
os.getcwd()
```

The *rmdir()* Method:

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax:

```
os.rmdir('dirname')
```

Example:

```
import os
os.rmdir( "/tmp/test" )
```

File & Directory Related Methods:

There are three important sources which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows:

- File Object Methods: The *file* object provides functions to manipulate files.
- OS Object Methods.: This provides methods to process files as well as directories.

Thank You!