Python for Informatics

LESSON 2

Copyright © 2015 Walter Wesley All Rights Reserved

More About Data Types

- In our previous lesson, we looked at mixed expressions that used operand values with integer and float types.
- There are four distinct **numeric** types: plain integers, long integers, floating point numbers, and complex numbers.
- These specific numeric types are often referred to by their shortened designations: *integer*, *long*, *float*, and *complex*.

Copyright © 2015 Walter Wesley All Rights Reserved

The Boolean Data Type

- "Boolean" is yet another data type, which manages values of either *true* or *false*.
- The *boolean* data type is used with logical control structures, and it is therefore a fundamental element of programming languages.
- A *boolean expression* is an expression that evaluates (or "resolves") to a boolean value of either *true* or *false*.

Copyright © 2015 Walter Wesley All Rights Reserved

The Boolean Data Type

In ascending priority, the *boolean* operations are:

Operators	Evaluated Result	Effect
x ***or*** y	if x is false, then y, else x	(1)
x ***and*** y	if x is false, then x, else y	(2)
not x	if x is false, then True, else False	(3)

Effect:

1.This is a short-circuit operator, so it only evaluates the second argument if the first one is *false*.

2.This is a short-circuit operator, so it only evaluates the second argument if the first one is *true*.

3. *not* has a lower priority than non-Boolean operators, so *not a == b* is interpreted as *not (a == b)*, and *a == not b* is a syntax error.

Copyright © 2015 Walter Wesley All Rights Reserved

Comparison Operators

- Comparison operators are also sometimes called *relational* operators—they evaluate how two operands compare or relate to each other.
- A *boolean expression* is an expression that evaluates (or "resolves") to a boolean value of either *true* or *false*.
- Since comparison or relational operators evaluate to either *true* or *false*, they are often used as elements of *boolean expressions*.

Copyright © 2015 Walter Wesley All Rights Reserved

Relational Operators

- "==" is the equality operator. It tests to see if the left operand and the right operand are equal, and evaluates to *true* if they are equal or *false* if they are not.

5 == 5
True

x = 8
x == 8
True

Copyright © 2015 Walter Wesley All Rights Reserved

Relational Operators

- Here is the list of relational operators in Python:

Operator	Meaning
“==”	Is Equal To
“!=”	Is Not Equal To
“>”	Is Greater Than
“<”	Is Less Than
“>=”	Is Greater Than or Equal To
“<=”	Is Less Than or Equal To

Copyright © 2015 Walter Wesley All Rights Reserved

Relational Operators

- “!=”

  x = 42
  x != 120
  True

- “>”

  x > 5
  True

- “<”

  x < 5
  False

- “>=”

  x >= 42
  True

Copyright © 2015 Walter Wesley All Rights Reserved

Common Programming Error!

- Important!
- "==" is not the same as "="
- "==" is the ***equals*** operator.
- "=" is the ***assignment*** operator.
- Interchanging one for the other can cause frustratingly subtle bugs that are difficult to identify.

Copyright © 2015 Walter Wesley All Rights Reserved

Conditional Execution – The "if" Statement

- Sometimes we want to execute something ***only*** when a certain condition applies.
- This is accomplished by means of an "if" statement.
- Here's an example of the syntax:

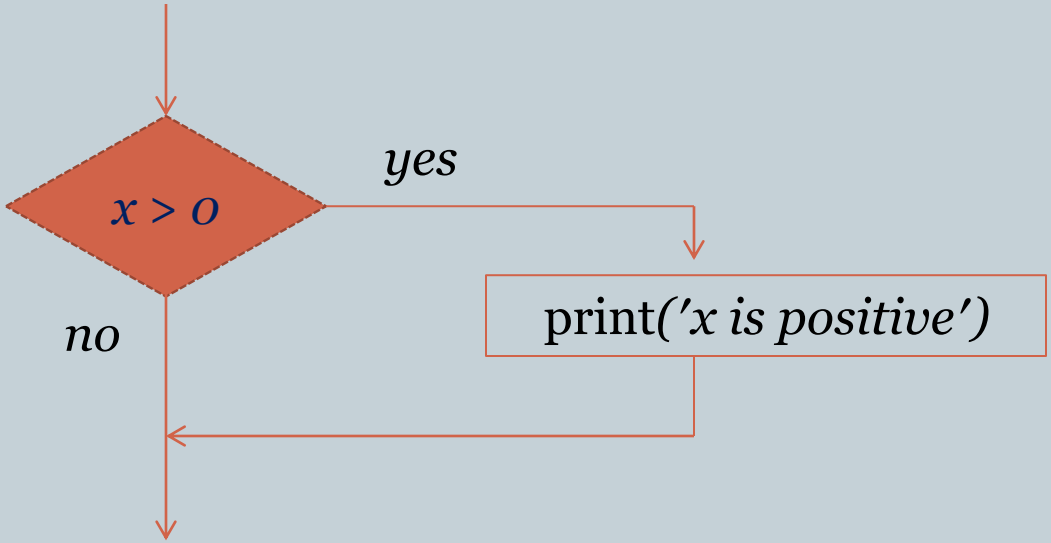
```
if x > 0 :
    print('x is positive')
```

- If the value of x is greater than zero, "x is positive" will be printed to the console.

Copyright © 2015 Walter Wesley All Rights Reserved

The "if" Statement Flow Chart

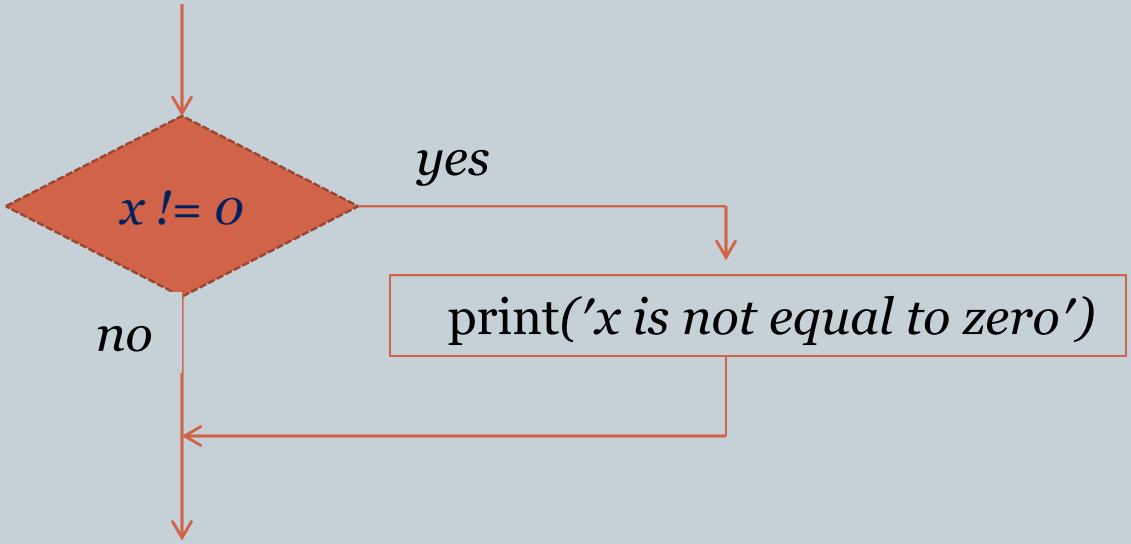
- Although flow charts are rarely used in software development, they are helpful when learning how conditional execution works.



Copyright © 2015 Walter Wesley All Rights Reserved

The "if" Statement Flow Chart

- All of the relational operators act exactly as you would expect.



Copyright © 2015 Walter Wesley All Rights Reserved

PEMDAS

- PEMDAS is a mnemonic that helps us remember the order of precedence of the various operations within a complex expression.
  - P for Parentheses: anything with parentheses is evaluated before anything else.
  - E for Exponentiation: This is the next highest precedence.
  - M for Multiplication
  - D for Division
  - A for Addition
  - S for Substraction: Any substraction is performed last!

Copyright © 2015 Walter Wesley All Rights Reserved

P

- Remembering the first letter, “P”, of the PEMDAS acronym is both sufficient and preferred.
- The aggressive use of parentheses ensures that you don’t need to interpret meaning of an expression by sorting through the PEMDAS mnemonic in your head.
- This is a case wherein redundancy is a good thing! Use parentheses to enforce the order of evaluation you want, even if the parentheses are not needed. This habit will substantially clarify your intent.

Copyright © 2015 Walter Wesley All Rights Reserved

Alternative Execution – The "if…else" Statement

- Sometimes we want to execute one thing when a certain condition applies, or another thing when the condition does not apply.
- This is accomplished by means of an "if…else" statement.
- Here's an example of the syntax:

```
if x > 0:
    print('x is positive')
else:
    print('x is not positive')
```

- If the value of x is greater than zero, "x is positive" will be printed, otherwise "x is not positive" will be printed.

Copyright © 2015 Walter Wesley All Rights Reserved

Nested Conditionals– Statement Layering

- Nested conditions allow use to construct successive questions as if they are russion dolls.
- Here's an example of the syntax:

```
if x > 0:
    print('x is positive')
else:
    if x < 0:
        print('x is not negative')
    else:
        print('x is zero')
```

Copyright © 2015 Walter Wesley All Rights Reserved

try/except Error Handling

- Error handling is a critical software feature.
- In order for software to be trustworthy, we must have confidence that it is correct, or that it will inform us when it is not correct.
- Without error handling, your program can crash in a surprising and confusing ways.
- The try/except construct enables your Python code to identify errors and respond to those errors in a graceful way.

Copyright © 2015 Walter Wesley All Rights Reserved

try/except Error Handling

- Execute the following code:

```
prompt = 'Please enter '

numerator = float(raw_input(prompt + 'the numerator:'))
denominator = float(raw_input(prompt + 'the denominator:'))

fraction = float(numerator) / denominator

print('The result of division is ' + str(fraction))
```

- For the first trial, use values 10 and 3.
- For the second trial, use values 10 and 0.

Copyright © 2015 Walter Wesley All Rights Reserved

Running tryExcept.py…

- Please enter the numerator:10
- Please enter the denominator:0
- ---
- ZeroDivisionError Traceback (most recent call last)
- C:\Users\SDCCD User\dev\pythonForInformatics\tryExcept.py in <module>()
- 4 denominator = float(raw_input(prompt + 'the denominator:'))
- 5
- ----> 6 fraction = float(numerator) / denominator
- 7
- 8 print('The result of division is ' + str(fraction))
- ZeroDivisionError: float division by zero

Copyright © 2015 Walter Wesley All Rights Reserved

Running tryExcept.py…


```
prompt = 'Please enter '

numerator = float(raw_input(prompt + 'the numerator:'))
denominator = float(raw_input(prompt + 'the denominator:'))

try:
    fraction = float(numerator) / denominator
    print('The result of division is ' + str(fraction))
except:
    print('The numerator cannot be zero!')
```

Copyright © 2015 Walter Wesley All Rights Reserved

Short-circuit Evaluation

- For reasons of efficiency, the Python interpreter will not evaluate a complex logical expression in its entirety, if it is not needed for deducing the final result.
- Given the expression… x > 10 and y > 100
  - If x is less than or equal to 10, the subexpression x > 100 will ***not*** be evaluated! It's not necessary, since we know that the full expression must be false regardless.

Copyright © 2015 Walter Wesley All Rights Reserved

The Guardian Pattern

- Short-circuit evaluation has given rise to the ***guardian pattern***.
- The *guardian pattern* is a coding technique that enables us to avoid computations that would otherwise be catastrophic.
- Instead of…

  ```
  x > 10 and x / y > 3
  ```

- With the *guardian pattern*, we have…

  ```
  x > 10 and y != 0 and x / y > 3
  ```

Copyright © 2015 Walter Wesley All Rights Reserved

Functions

- A *function* in Computer Science is based upon the mathematical concept of a function.
- There are three differences between a mathematical function and a program function.
  1. While a math function requires an argument, a program function does not.
  2. While a math function necessarily returns a value, some program functions do not.
  3. While math functions are performed by human brains, program functions are performed by computation machines.

Copyright © 2015 Walter Wesley All Rights Reserved

Functions

- A *function* is a named sequence of executable statements.
- The name that is specified when the function is defined provides a means of *invoking* or *calling* that function.
- *User-defined functions* are functions that are defined by a programmer to perform a desired computation.
- Built-in functions are functions that are predefined as part of the core Python language.
- Having a library of predefined functions is an extremely powerful programming resource.

Copyright © 2015 Walter Wesley All Rights Reserved

Built-in Functions

- As examples of built-in functions, let us look at the ***max*** and ***min*** functions.
- These functions take a ***sequence*** argument, and return a value that corresponds to the associated item within the sequence.
- ***max*** returns the maximum value within the sequence, and ***min*** returns the minimum value within the sequence.
- The items within a string sequence are compared based on their alphabetical order.

```
max('spam')
's'

min('spam')
'a'
```

- ***max*** and ***min*** are built-in and ready to use whenever they are needed.

Copyright © 2015 Walter Wesley All Rights Reserved

Built-in Functions

- ***max*** and ***min*** will work with any type of sequence. It just depends on what it means for one list item to be greater than or less than another item.

max(16, 5, 9, 10, 23, 7)

23

min(16, 5, 9, 10, 23, 7)

5

Copyright © 2015 Walter Wesley All Rights Reserved

Built-in Functions

- Another very useful function is the ***len*** function, which returns the length (i.e., number of items) of an object. The argument can be a *sequence* (such as a *string* or *list*) or a *collection*.
- We will explore *collections* later.
- Here is an exampe of using the len built-in function:

```
len('Monty')
5
```

Copyright © 2015 Walter Wesley All Rights Reserved

Lists vs Sequences

- A Python ***list*** is an ordered sequence of items that is referred to by a single ***name***, such that any given item within the list can be specified by using the ***list*** name along with an ***index*** value that ***selects*** the desired item.
- The indexing is ***zero-based***, which means that 0 selects the first item, 1 selects the second item, etc.

```
n = [5, 30, 62, 11, 42]
print(n[0])
5
```

```
s = 'spam'
print(s[2])
'a'
```

- What Python calls a ***list*** is called an ***array*** is most other languages.

Copyright © 2015 Walter Wesley All Rights Reserved

Lists vs Sequences

- All lists are sequences, but not all sequences are lists.
- The seven sequence types are:
  - strings
  - unicode strings
  - lists
  - tuples
  - bytearrays
  - buffers
  - xrange objects.
- Lists are constructed by means of square brackets.

```
n = [5, 30, 62, 11, 42]
```

Copyright © 2015 Walter Wesley All Rights Reserved

Mutable vs Immutable Sequences

- ***list*** is an example of a ***mutable*** sequence, which means that the items within it ***can be changed***.

```
n = [5, 30, 62, 11, 42]
print(n[3])
11

n[3] = 77

print(n[3])
77
```

Copyright © 2015 Walter Wesley All Rights Reserved

Mutable vs Immutable Sequences

- By contrast, ***string*** is an example of an ***immutable*** sequence, which means that the items within it ***cannot be changed***.

```
s = 'spam'
print s[2]
a

s[2] = 'i'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-3728c4c73307> in <module>()
----> 1 s[2] = 'i'

TypeError: 'str' object does not support item assignment
```

Copyright © 2015 Walter Wesley All Rights Reserved

Type Conversion Functions

- Python also provides built-in functions that convert values of one type to values of another.
- For example, if we need to perform a type conversion from a *string* value to an *integer* value, we do the following:

```
ans = '42'

int(ans)
42

print(int(ans) / 2)
21
```

Copyright © 2015 Walter Wesley All Rights Reserved

Type Conversion Functions

- Of course, we can only perform conversions when a sensible conversion is possible.

```
name = 'Roger the Shrubber'

int(name)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-30-3627f499edfd> in <module>()
----> 1 int(name)

ValueError: invalid literal for int() with base 10: 'Roger the Shrubber'
```

Copyright © 2015 Walter Wesley All Rights Reserved

The *random* Function

- Although computers do not understand randomness, and cannot provide true random numbers, there are many scenarios (games, for example) in which simulating randomness is especially useful.
- Numbers that made to appear random with computers are called ***pseudorandom numbers***.
- The ***random* function** is a built-in function that returns pseudorandom float values.
- The ***random*** function is one of many functions within the ***random module*** that supports dealing with randomness features.

Copyright © 2015 Walter Wesley All Rights Reserved

The *random* Function

- In order to use any of the randomness features, the random module must be imported.

```
import random

for i in range(5):
    num = random.random()
    print(num)
```

```
0.201086994053
0.176631734158
0.305493930365
0.582269675033
0.143482947831
```

Copyright © 2015 Walter Wesley All Rights Reserved

The *random* Function

- ***randint()*** is another random function that takes low and high parameters as the low and high range bounds of the random integer values that it returns.

```
import random

for i in range(5):
    num = random.randint(80, 85)
    print(num)
```

```
81
81
84
80
85
```

Copyright © 2015 Walter Wesley All Rights Reserved

The *choice* Function

- There's even a way that you can make pseudorandom selections or *choices* from a given sequence of items.

```
import random

door_numbers = ['one', 'two', 'three']
my_pick = random.choice(door_numbers)
print(my_pick)

three
```

Copyright © 2015 Walter Wesley All Rights Reserved

Math Functions

- The Python ***math*** module provides an assortment of useful math functions.
- When you import a module, you are creating a module object with that name.
- If you print the module object, you will see information about the module, such as whether it is a built-in module, and it if not a built-in, from what location it was loaded:

```
import random
import math

print(math)
<module 'math' (built-in)>

print(random)
<module 'random' from
'C:\Users\SDCCD_User\AppData\Local\Enthought\Canopy\App\appdata\canopy-
1.5.2.2785.win-x86_64\lib\random.pyc'>
```

Copyright © 2015 Walter Wesley All Rights Reserved

Math Functions

- Let's look at just a few of the functions available in the ***math*** module:

```
import math

signal_power = 50.1
noise_power = 1.7

ratio = signal_power / noise_power

decibels = 10 * math.log10(ratio)

print(decibels)
14.6938880449
```

- Notice how the variable names use an underscore to separate multiple words with the name (e.g., *signal_power*).

Copyright © 2015 Walter Wesley All Rights Reserved

Math Functions


```
import math

degrees = 30
radians = degrees / 360.0 * 2 * math.pi

sine_value = math.sin(radians)

print(sine_value)
0.5
```

- Notice the use of ***dot notation***. The module name (‘***math’)*** is followed immediately by a dot (‘.’), and then by the function name (‘***sin***’).

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- You can define your own functions!
- Here is the function definition syntax:

```
def function_name():
    statement1
    statement2
    .
    .
    .
```

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- Here is a simple example:

```
def dead_parrot():
    print('''He's not pinin'!''')
    print('''He's passed on!''')
    print('''This parrot is no more!''')
    print('''He has ceased to be!''')
```

- Notice the use of the triple single quotes (''') that allow for the embedding of single quotes within the string.

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- The first line of the function definition is called the ***header***:

```
def  dead_parrot():
```

- The rest of the function is called the ***body***:

```
print("'He's not pinin'!"')
print("'He's passed on!"')
print("'This parrot is no more!"')
print("'He has ceased to be!"')
```

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- Defining a function creates a variable with the same name.
- You can ***print*** this variable:

```
print(dead_parrot)
<function dead_parrot at 0x0000000009AEB898>
```

- You can verify the type of the variable with the ***type*** function.

```
print(type(dead_parrot))
<type 'function'>
```

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- To call a defined function, just type in the name (including the parentheses):

dead_parrot()

He's not pinin'!

He's passed on!

This parrot is no more!

He has ceased to be!

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- To call a defined function, just type in the name (including the parentheses):

dead_parrot()
He's not pinin'!
He's passed on!
This parrot is no more!
He has ceased to be!

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- Functions serve as powerful *abstractions*.
- User-defined functions allow us to represent ***ways of doing things***.
- Functions can be called again and again, from different contexts.
- Functions provide an important kind of ***software reuse***.

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- Functions can be defined to call other functions.
- This allows for an important technique known as ***functional decomposition***.
- Through functional decomposition, you can ***take a complex process and break it down*** into a set of simpler functions.
- By repeatedly breaking things down into smaller and smaller pieces, you ultimately arrive at a granularity where things are easy to build and assemble.
- By factoring code out into smaller pieces, you can avoid duplicating code unnecessarily.

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

- As a simple example of a function calling another function, let's define a dead_parrot_refrain() function that calls the dead_parrot() function twice.

```
def dead_parrot_refrain():
    dead_parrot()
    dead_parrot()
```

- Now when we call this we get…

Copyright © 2015 Walter Wesley All Rights Reserved

User-defined Functions

dead_parrot_refrain()
He's not pinin'!
He's passed on!
This parrot is no more!
He has ceased to be!
He's not pinin'!
He's passed on!
This parrot is no more!
He has ceased to be!

Copyright © 2015 Walter Wesley All Rights Reserved

Functions Execution Flow

- It is important to understand that a function call alters the execution flow of your program, in that the call is a sort of detour.
- Normal flow executes one statement after another, step by step.
- But a function call causes the flow to jump to the body of the function, each statement of the body is sequentially executed, and only then does the flow return to the point at which the function was called.

Copyright © 2015 Walter Wesley All Rights Reserved

Functions Parameters

- Defining a function that uses parameters is easy:

```
def echo(something_to_say):
    print(something_to_say)
    print(something_to_say)
    print(something_to_say)


echo('The Larch!')
The Larch!
The Larch!
The Larch!
```

Copyright © 2015 Walter Wesley All Rights Reserved

Functions that Return Void

- The ***echo*** function that we defined is an example of a ***void function***.
- *void functions* are functions that don't return anything.

Copyright © 2015 Walter Wesley All Rights Reserved

Functions with Return Values

- Defining a function that uses a parameter ***and*** returns a value takes a little more work:

```
def  half(some_number):
    return some_number / 2.0

print(half(7))
3.5
```

Copyright © 2015 Walter Wesley All Rights Reserved

Functions with Local Variables

- Functions can use variables that are local to their body's scope:

```
def average_three_numbers(n1, n2, n3):
    total = n1 + n2 + n3
    return total / 3.0

print(average_three_numbers(3, 5, 7))
4.33333333333
```

Copyright © 2015 Walter Wesley All Rights Reserved

Functions with Local Variables

- The variable total is local to the scope of the average_three_numbers() function, which means that it comes into existence when it is first referenced, and it goes out of existence when the execution flow leaves the function.
- After the average_three_numbers() function is called, attempting to reference the local variable causes an error:

```
print(total)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-90-bc4e21da9abf> in <module>()
----> 1 print(total)

NameError: name 'total' is not defined
```

Copyright © 2015 Walter Wesley All Rights Reserved