COMP47670 (COMP41680)

Introduction to Python
Part 2

Slides by Derek Greene

UCD School of Computer Science

Spring 2019

Overview

- Commenting
- Defining Functions
  - Return Values
  - Function Composition & Recursion
- Variable Scope
- Working with Strings
- Dynamic Typing
- Converting Between Types
- String Formatting

Commenting Code

- Comments provide a way to write human-readable documentation for your code. Key part of programming!
- In Python code, anything after a # and continuing to the end of the line is considered to be a comment and is ignored.

```
x = 5*4 # ignore this
```

```
x = 5 + 3 # + 10
```

- Multi-line comments can also be added to Python code, using triple quoted strings (i.e. 3 single or 3 double quote characters):

```
'''
This is a single quoted
multi-line comment.
'''
```

```
"""
This is a double quoted
multi-line comment.
"""
```

- Note: if you are inside an indented block of code, multi-line comments need to be indented too! Not the case for # comments.

Functions in Python

- Functions in Python represent a block of reusable code to perform a specific task.
- Two basic types of functions:
  - Built-in functions: these usually a part of existing Python packages and libraries.
  - User-defined functions: written by programmers to meet certain requirements of a task or project.
- User-defined functions only need to be written once, and can then potentially be reused multiple times in different applications. They provide a means of making your code more organised and easier to maintain.

Defining Functions

- We create a new user-defined function in Python using the def keyword, followed by a block of code. Specifically we need:
  1. A function name
  2. Zero or more input arguments
  3. An optional output value, specified via return keyword
  4. A block of code

Function definition must start with `def`

Argument names, in parenthesis

... and end with a colon

```
def subtract(x, y):
    return x - y
```

Block of code

- Call the new function using parenthesis notation:

```
z = subtract(5,3)
```

```
z = subtract(8,12)
```

Defining Functions

- In the simplest case, we can define a function that does not take any input. More often, we will want to pass values to a function as input arguments.

```
def sayhello():
    print("hello!")
```

```
sayhello()
```

```
hello!
```

```
def add(x, y):
    print( x + y )
```

```
add(3, 5)
```

```
8
```

- Some arguments for a function can be optional and do not need to be specified if we provide a default value:

```
def add(x, y=1):
    print( x + y )
```

```
add(3, 5)
```

```
8
```

```
add(3)
```

```
4
```

If a value for argument y is not specified, the default value will be y=1

Calling Functions

- Functions are run when you call them with parenthesis notation:

Function name

Argument values, in parenthesis

Return value

```
res = pow(4, 3)
```

e.g. call function `pow()` to calculate 4 raised to the power of 3

- We can also use keyword arguments that are specified by name.
- Example: One available keyword argument for `print()` is `sep`, which specified what characters should be used to separate multiple values:

```
print(5, 10, 15)
```

```
5 10 15
```

```
print(5, 10, 15, sep="_")
```

```
5_10_15
```

- When non-keyword arguments are used together with standard keyword arguments, keyword arguments must come at the end.

Returning Values

- The type of value returned by a function does not need to be specified in advance.
- Often it is useful to have multiple return statements, one in each branch of a conditional.
- Code that appears after a return statement cannot be reached and will never be executed.

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
    return 0
```

Code will never run

- If no return value is specified, a function will return None by default.

```
def square( x ):
    y = x * x
```

```
res = square( 3 )
print(res)
```

```
None
```

Returning Values

- Python allows multiple values to be returned from a single function by separating the values with commas in the return statement.
- Multiple values get returned as a tuple.

```
def min_and_max(values):
    vmin = min(values)
    vmax = max(values)
    return vmin, vmax
```

Two values returned

```
values = [5, 19, 3, 11, 24]
result = min_and_max(values)
print(result)
```

```
(3, 24)
```

Result is a tuple with 2 values

- Unpacking: Multiple variables can be assigned the multiple values returned by the function in a single statement.

```
x, y = min_and_max(values)
print(x)
print(y)
```

Put the 1st returned value in `x`
Put the 2nd returned value in `y`

```
3
24
```

Defining Functions: Examples

- Functions for Celsius to Fahrenheit conversion, and vice-versa:

```
def celsius_to_fahrenheit(c):
   return (9.0/5.0 * c) + 32
```

```
def fahrenheit_to_celsius(f):
   return (f - 32.0) * 5.0 / 9.0
```

```
for ctemp in range(0,30,5):
   print("Celsius", ctemp)
   ftemp = celsius_to_fahrenheit(ctemp)
   print("Fahrenheit", ftemp)
```

```
for ftemp in range(50,80,5):
   print("Fahrenheit", ftemp)
   ctemp = fahrenheit_to_celsius(ftemp)
   print("Celsius", ctemp)
```

```
Celsius 0
Fahrenheit 32.0
Celsius 5
Fahrenheit 41.0
Celsius 10
Fahrenheit 50.0
Celsius 15
Fahrenheit 59.0
Celsius 20
Fahrenheit 68.0
Celsius 25
Fahrenheit 77.0
```

```
Fahrenheit 50
Celsius 10.0
Fahrenheit 55
Celsius 12.777777777777779
Fahrenheit 60
Celsius 15.555555555555555
Fahrenheit 65
Celsius 18.333333333333332
Fahrenheit 70
Celsius 21.11111111111111
Fahrenheit 75
Celsius 23.88888888888889
```

Defining Functions: Examples

- Function for finding the Least Common Multiple (LCM) of two numbers. That is, the smallest positive integer that is perfectly divisible by the two given numbers.

Define function lcm with 2 input parameters

Use while loop to test increasingly larger values

Return value found to be the LCM of x and y

```
def lcm(x, y):
  # choose the greater number
  if x > y:
    greater = x
  else:
    greater = y
  # keep increasing until we find the answer
  while True:
    if(greater % x == 0) and (greater % y == 0):
      answer = greater
      break
    greater += 1
  return answer
```

Call our new function

```
print( lcm(5,7) )
```

```
35
```

```
print( lcm(12,30) )
```

```
60
```

Function Composition & Recursion

- You can call one function from inside another. Several simple functions can be combined to create more complex ones.

```
def square(x):
    return x*x
```

```
def negative(x):
    return -x
```

```
def calc_score(x, y):
    a = square(x)
    b = negative(y)
    return a + b
```

```
calc_score( 3, 4 )
```

```
5
```

- Recursive functions repeatedly call themselves either directly or indirectly in order to loop.

```
def mysum( l ):
    if len(l)==0:
        return 0
    return l[0] + mysum(l[1:])
```

```
mysum( [1, 2, 3] )
```

```
6
```

Example recursively sums a list of numbers. What's actually happening here:

```
mysum([1, 2, 3])
mysum([2, 3])
mysum([3])
mysum([])
```

Variable Scope

- Scope: refers to the ability to access certain variables in a certain part of our code.
- Code written at the top level (i.e. not in a nested block) is global. These variables are accessible everywhere. Variables defined in a function are local, and are accessible only in that function.

This variable is **global**, it is accessible everywhere

This variable is **local**, it is only accessible in its own function

```
gvar = "This is global"

def myfunction():
  lvar = "This is local"
  print("global_var:", global_var)
  print("lvar:", lvar)
```

```
myfunction()
```

```
gvar: This is global
lvar: This is local
```

We cannot access the local variable outside the function

```
print("gvar:", gvar)
print("lvar:", lvar)
```

```
gvar:  This is global
NameError: name 'lvar' is not defined
```

Variable Scope

- Global variables can be accessed in a function, but normally cannot be modified.

```
x = 20

def myfunction():
    x = 15

myfunction()
print(x)
```

```
20
```

We are creating a new local variable with the same name!

No change to the original!

- We can use the `global` statement to tell Python that a function plans to change one or more global variables.

```
x = 20

def myfunction():
    global x
    x = 15

myfunction()
print(x)
```

```
15
```

Tell Python we plan to modify the global variable `x` in this function

Value of global variable has been changed

Strings Revisited

- Recall Python strings can be defined using either single or double quotes.
- Python also has block strings for multi-line text, defined using triple quotes (single or double).
- Escape sequences: backslashes are used to introduce special characters.

Escape	Meaning
`\n`	Newline character
`\t`	Tab character
`\r`	Return character (Windows)
`\\`	Backslash - same as one '\'

```
mytext = "this is some text"
```

```
mytext = 'this is some text'
```

```
s = """School of CS,
UCD,
Belfield"""
```

```
s
```

```
'School of CS,\nUCD,\nBelfield'
```

```
address = "UCD\tBelfield"
address
```

```
'UCD\tBelfield'
```

```
address = "UCD\tBelfield"
print(address)
```

```
UCD	Belfield
```

Working With Strings

- Strings can be viewed as sequences of characters of length N.

```
s = "BELFIELD"
```

0	1	2	3	4	5	6	7
B	E	L	F	I	E	L	D

- As such, we can apply many standard list operations and functions to Python strings.

```
s[2]
```

```
L
```

Access a character by index (position)

```
s[1:4]
```

```
ELF
```

Create substrings via slicing

```
len(s)
```

```
8
```

Length of the string i.e. number of characters

- Characters and substrings can be accessed using square bracket notation just like lists.

- Strings can be concatenated together using + operator

```
t = "ucd" + "_" + "belfield"
t
```

```
'ucd_belfield'
```

String Functions

- Strings have associated functions to perform basic operations.

Syntax
`<string_variable>.<function>(argument1, argument2, ...)`

- Example of string manipulation functions - case conversion:

```
s = "Hello World"
s.upper()
```

```
'HELLO WORLD'
```

```
s = "Hello World"
s.lower()
```

```
'hello world'
```

```
s = "Hello World"
s.swapcase()
```

```
'hELLO wORLD'
```

```
s = "Hello World"
t = s.upper()
print(s)
```

```
'Hello World'
```

```
print(t)
```

```
'HELLO WORLD'
```

These string manipulation functions make a copy of the original string, they do not change the original string.

String Functions - Find & Replace

- Strings have associated functions for finding characters or substrings.

Search for the first occurrence of the specified substring.

```
s = "Hello World"
s.find("World")
```

```
6
```

Returns either the index of the substring, or -1 if not found.

```
s.find("UCD")
```

```
-1
```

Count number of times a substring appears in a string.

```
x = "ACGTACGT"
x.count("T")
```

```
2
```

```
x = "ACGTACGT"
x.count("U")
```

```
0
```

- We can also replace characters or complete substrings. This creates a new copy of the original string.

```
x = "ACGTACGT"
x.replace("T","V")
```

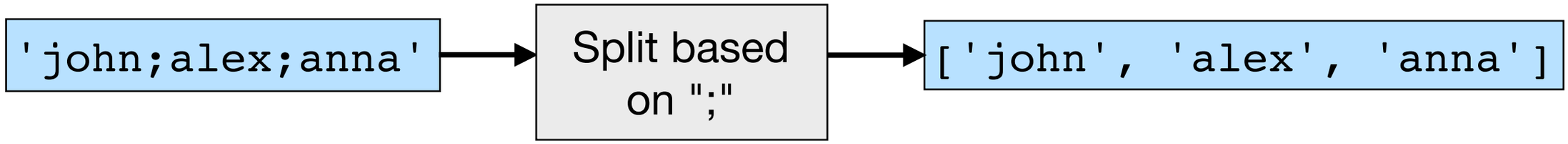
```
'ACGVACGV'
```

```
x = "Hello World"
x.replace(" ","_")
```

```
'Hello_World'
```

String Functions - Split & Join

- Use the `split()` function to separate a string into multiple parts, based on a delimiter - i.e a separator character or substring.



Output is a list containing multiple string values

```
names="john;alex;anna"
names.split(";")
```

```
['john', 'alex', 'anna']
```

```
data = "5,6,11,12"
data.split(",")
```

```
['5', '6', '11', '12']
```

- Use the `join()` function to concatenate a list of strings into a single new string. All values in the list must be strings.

```
<separator>.join(list)
```

```
l = ["dublin","cork","galway"]
"$".join(l)
```

```
'dublin$cork$galway'
```

Dynamic Typing

- Python uses a dynamic typing model for variables:
  - Variables do not need to be declared in advance.
  - Variables do not have a type associated with them, values do.

```
x = 2
x = "some text"
x = True
```

We can change the type of a variable by simply assigning it a new value

- Python uses strong dynamic typing
  - Applying operations to incompatible types is not permitted.
  - May need to remember the type of value your variables contain!

```
1 + "hello"
```

```
Traceback (most recent call last):
  File "", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and ‘str'
```

Cannot add an integer to a string!

Converting Between Types

- Since mixing incompatible types is not permitted, we use built-in conversion functions to change a value between basic types.

Use the `str()` function to convert any value to a string

```
str(27)
```

```
'27'
```

```
str(0.45)
```

```
'0.45'
```

We can also convert strings to numeric values using `int()` and `float()`

```
s = "145"
int(s)
```

```
145
```

```
s = "1.325"
float(s)
```

```
1.325
```

- Not all strings can be converted to numeric values…

```
int("UCD")
```

```
ValueError: invalid literal for int() with base 10: 'UCD'
```

```
float("ax0.353")
```

```
ValueError: could not convert string to float: 'ax0.353'
```

Converting Between Types

- Often use the string `split()` function in conjunction with type conversion when parsing simple data files...

```
data = "0.19,1.3,4.5,3,12"
parts = data.split(",")
print( parts )
```

```
['0.19', '1.3', '4.5', '3', '12']
```

Call `split()` to divide the original string into a list of strings

```
values = []
for s in parts:
    values.append( float(s) )
```

Convert each sub-string to a float value

```
print( values )
```

```
[0.19, 1.3, 4.5, 3.0, 12.0]
```

```
type( values[0] )
```

```
<class 'float'>
```

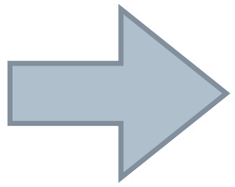
String Formatting

- In Python, we can concatenate multiple variables of different types into a single string, using the % operator. The format string provides the recipe to build the string, containing zero or more placeholders.

Syntax
`"<format string>" % (<var1>,<var2>,...,<varN>)`

- The placeholders get substituted for the list of values that you provide after the % symbol. The number of placeholders in the format string must equal the number of values!

```
" %s was born in %s in %d " % ( "John", "Dublin", 1985 )
```



```
" John was born in Dublin in 1985 "
```

String Formatting

- Special placeholder codes are used when building a format string.
- Each placeholder should correspond to the type of the value that will replace it.

Building format strings

Code	Variable Type
`%d`	Integer
`%f`	Floating point
`%.Nf`	Float (*N* decimal places)
`%s`	String (or any value)
`%%`	The '%' symbol

`\t`	Tab character
`\n`	Newline character

```
x = 45
y = 0.34353
z = "text"
s = "%d and %.2f and some %s" % (x,y,z)
print( s )
```

```
'45 and 0.34 and some text'
```

```
s2 = "%f => %.0f or %.4f" % (y,y,y)
print( s2 )
```

```
0.343530 => 0 or 0.3435
```