COMP41680

Next Steps in Python

Derek Greene

UCD School of Computer Science

Spring 2023

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 our 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

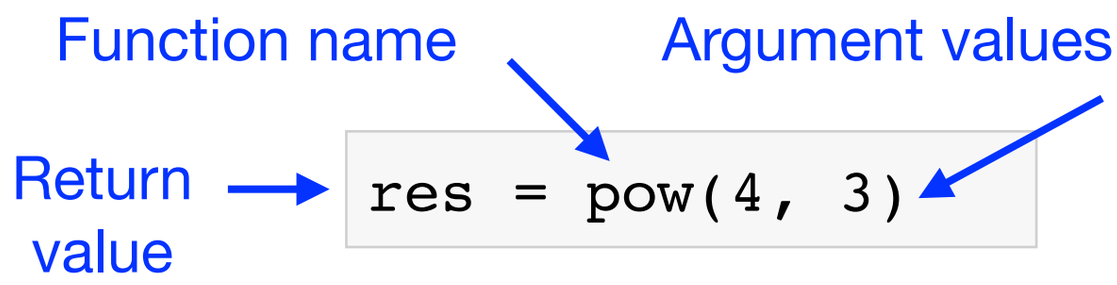
- Call the new function using parenthesis notation:

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

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

Calling Functions

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



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, rather than by position.
- Example: One available keyword argument for `print()` is `sep`, which specifies the characters to use to separate multiple values.

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

```
5 10 15
```

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

```
5_10_15
```

- When standard positional arguments are used together with 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
```

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)
```

```
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
```

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

```
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
```

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.

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

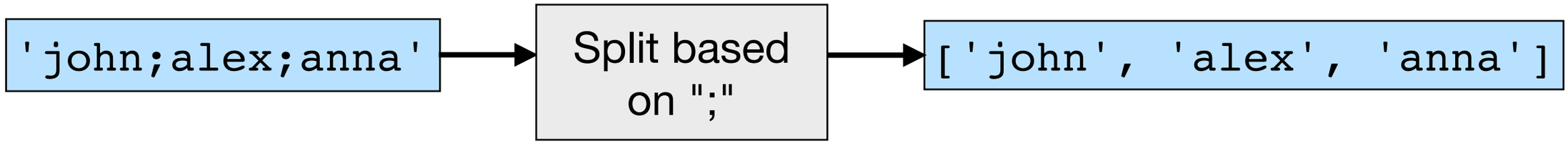
```
'ACGVACGV'
```

```
z = "Hello World"
z.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 our 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'>
```

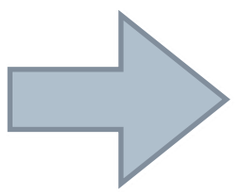
String Formatting

- 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 we 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
```

File Input/Output

- Files are special types of variables in Python, which are created using the `open()` function. Remember to `close()` the file when you are finished!

```
f = open( "<filepath>", "<action>" )
      .....
f.close()
```

"r": read
"w": write
(default is read)

- **Reading files:** After opening a file to read, you can use several functions to access plain-text data:

`read()`	read the full file
`readline()`	read a full line from a file
`readlines()`	read all lines from a file into a list

Need to strip line endings

```
f = open("test.txt", "r")
lines = f.readlines()
f.close()
for line in lines:
    line = line.strip()
    print(line)
```

Read all lines from a file into a list

Example: Reading Files

- Read a list of names and student numbers, storing the information in a dictionary.

Input: students.txt

```
17211426,Stephanie Gale
16212133,Jill Doyle
13388136,Pat Gilbert
17211824,Daryl Bishop
16216364,Carlos Alvarado
17211833,Alison Rogers
17212834,Neil Smith
13312141,Sandra Wright
```

```
register = {}
fin = open("students.txt","r")
lines = fin.readlines()
fin.close()
for line in lines:
   line = line.strip()
   parts = line.split(",")
   student_id = int(parts[0])
   fullname = parts[1]
   register[student_id] = fullname
```

Need to strip line endings

- Display the new contents of the dictionary:

```
for sid in register:
   print( "%d -> %s" % (sid, register[sid]) )
```

```
17211426 -> Stephanie Gale
16212133 -> Jill Doyle
13388136 -> Pat Gilbert
17211824 -> Daryl Bishop
16216364 -> Carlos Alvarado
17211833 -> Alison Rogers
17212834 -> Neil Smith
13312141 -> Sandra Wright
```

Working with Files

- **Writing files:** After opening a file to write, use the `write()` function to output strings to the file.

```
names = ["Mark","Lisa","Alice","Bob"]
f = open("out.txt","w")
for name in names:
   f.write(name)
   f.write("\n")
f.close()
```

Need to explicitly move to next line

out.txt

```
Mark
Lisa
Alice
Bob
```

- Note: By default Python will overwrite an existing file with the same name if it already exists.
- To add data to the end of an existing file, use append mode "a" when opening the file:

```
f = open("out.txt","a")
```

Indicates open in append mode

Example: Writing Files

- Read a list of lines from one file, write the contents back out to a second file with an additional prefix.

Open two files: one to read ("r"), one to write ("w")

```
fin = open("sample.txt","r")
fout = open("modified.txt","w")
for line in fin.readlines():
   fout.write("Copy: ")
   fout.write(line)
fin.close()
fout.close()
```

Note that the lines already end with a new line character

Input: sample.txt

```
County Dublin
County Galway
County Limerick
County Louth
County Wexford
```

Output: modified.txt

```
Copy: County Dublin
Copy: County Galway
Copy: County Limerick
Copy: County Louth
Copy: County Wexford
```

Python Error Messages

- A key programming task is debugging when a program does not work correctly or as expected.
- If Python finds an error in your code, it raises an exception.
  e.g. We try to convert incompatible types
  e.g. We try to read a non-existent file
  Also... When we have invalid syntax in our code (a "typo")

```
number = int("UCD")
```

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    number = int("UCD")
ValueError: invalid literal for int() with base 10: 'UCD'
```

Where the error occurred

Type of exception that has occurred

Text describing the error

Python Error Messages

```
d = {"Ireland": "Dublin"}
d["France"]
```

```
Traceback (most recent call last):
  File "test2.py", line 2, in <module>
    d["France"]
KeyError: 'France'
```

Where the error occurred

Type of exception that has occurred

```
def showuser(username):
  print(user_name)

showuser("bob")
```

```
Traceback (most recent call last):
  File "test3.py", line 4, in <module>
    showuser("bob")
  File "test3.py", line 2, in showuser
    print(user_name)
NameError: name 'user_name' is not defined
```

Error originated here

Exception Handling

- By default, an exception will terminate a script or notebook.
- We can handle errors in a structured way by "catching" exceptions. We plan in advance for errors that might occur...

General Format

```
try:
   <block of code>
except:
   <error handling block>
```

or

General Format

```
try:
   <block of code>
except <errorType>:
   <error handling block>
```

```
try:
    f = open("file.txt","r")
except:
    print("Some error occurred")
```

Code where error might occur

Handle any type of error

```
try:
    f = open("file.txt","r")
except IOError:
    print("Input/Output error")
```

Handle specific type of error

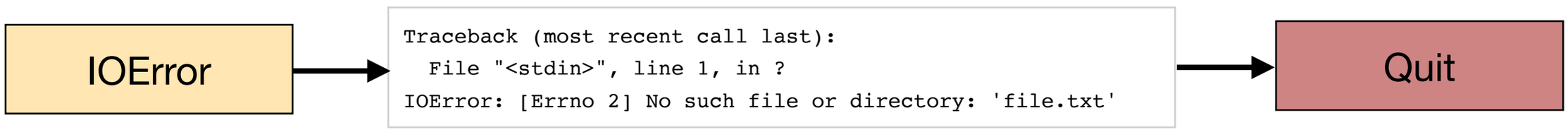
Exception Handling

- With exception handling: When an exception occurs, the program will continue, as well as inform you about the fact that the file read operation was not successful.

```
f = open("file.txt","r")
text = f.read()
f.close()
```

```
try:
    f = open("file.txt","r")
    text = f.read()
    f.close()
except IOError:
    print("Input/Output error")
```

Without exception handling



With exception handling

Python Modules

- Module: A single file of Python code, typically containing function and variable definitions related to a particular programming task.
- We can access the definitions in a module using the `import` keyword. There are two different ways to do this.
- The simplest approach is to `import` a whole module in its entirety:

```
import math
import sys
import sys, os
```

Import whole modules. Note we can import multiple modules on each line.

- The second approach is to import specific names from a module without importing the entire module by using the `from...import` functionality:

```
from sys import exit
from math import pi, e
```

Only import the functions or variables we require, do not import anything else.

Python Modules

- If we have imported an entire module, we can access its functions and variables using "dot notation" - i.e. the name of the module followed by the dot operator:

Import module and access a function

```
import math
x = math.sqrt(9)
```

If we forget to import the module...

```
x = math.sqrt(9)
```

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

- If we have imported a subset of a module, we can access all of those functions or variables without using dot notation - i.e. we do not require the module name as a prefix.

```
from math import sqrt, log
x = sqrt(9)
y = log(2)
print(x+y)
```

```
3.693147180559454
```

Now if we include the prefix in the call, it won't work...

```
x = math.sqrt(9)
```

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

Python Module Aliases

- In some cases we might import a module by giving it a shorter "alias" to save typing time. This is done using `import...as`. We can then use the new alias with dot notation as before.

Syntax
`import <module_name> as <short_name>`

```
import math
x = math.sqrt(9)
```

```
import math as m
x = m.sqrt(9)
```

Both code examples will produce the same result

```
import pandas as pd
p = pd.Series([1,2,3,4,5])
```

We will see many examples of using module aliases during the course

```
import numpy as np
v = np.array([1.0,2.0,3.0])
```