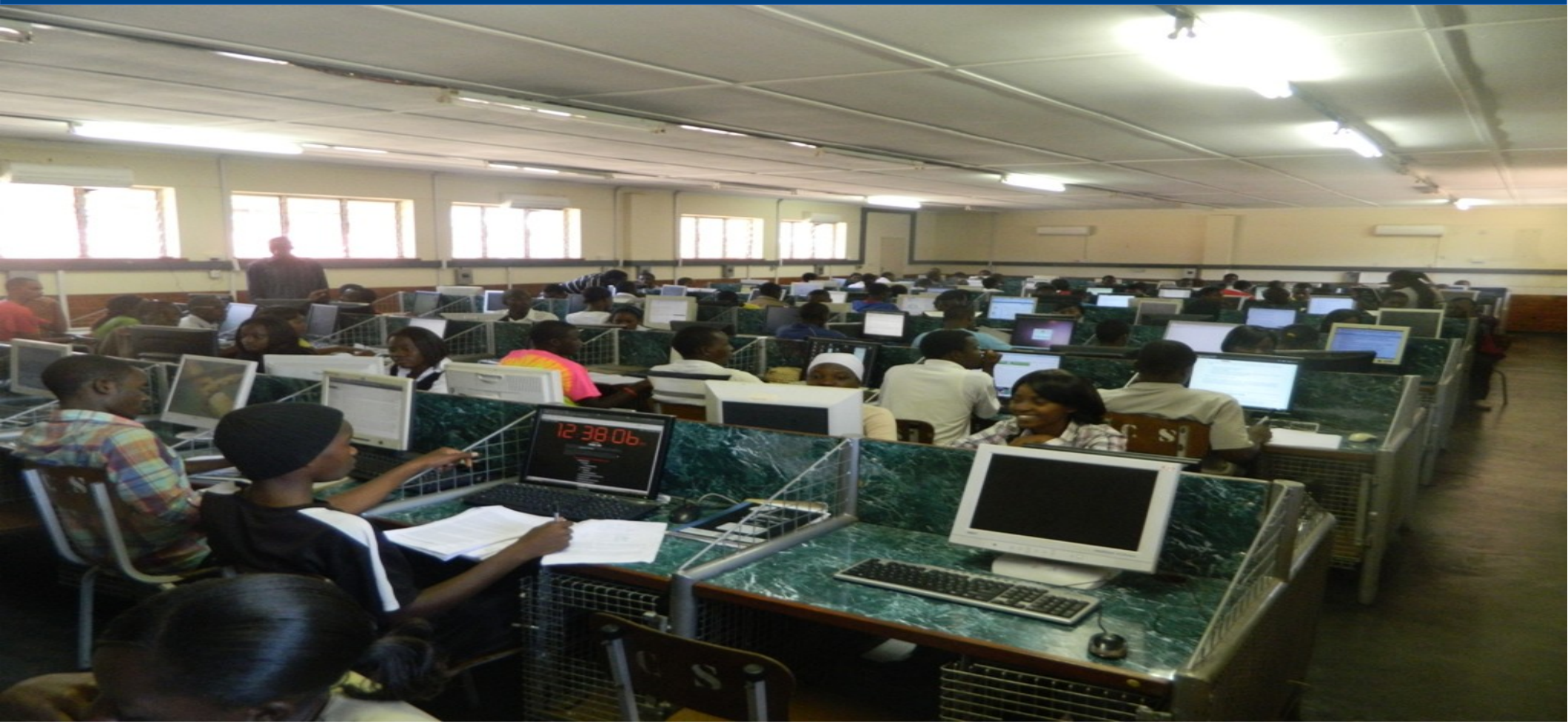
OOP



ICT & Electronics

Introduction to Python Object-Oriented Programming

T D KAVU

Modules and packages

- For small programs, we can just put all our classes into one file and put some code at the end of the file to start them interacting.
- However, as our projects grow, it can become difficult to find one class that needs to be edited among the many classes we've defined. This is where **modules** come in.
- **Modules** are simply **Python files**, nothing more
- If we have two files in the same folder, we can load a class from one module for use in the other module
- For example, if we are building an e-commerce system, we will likely be storing a lot of data in a database.We can put all the classes and functions related to database access into a separate file (we'll call it something sensible: database.py)
- Then our other modules (for example: customer models, product information, and inventory) can import classes from that module in order to access the database.
- The import statement is used for importing modules or specific classes or functions from modules.
- We used the import statement to get Python's built-in math module so we could use its sqrt function in our distance calculation.

Continue

- Assume we have a module called database.py that contains a class called Database , and a second module called products.py that is responsible for product-related queries.
- What we know is that products.py needs to instantiate the Database class from database.py so it can execute queries on the product table in the database.
- There are several variations on the import statement syntax that can be used to access the class.
- *import database*
- *db = database.Database()*
- This version imports the database module into the products *namespace* (the list of names currently accessible in a module or function), so any class or function in the database module can be accessed using database.<something> notation.
- Alternatively, we can import just the one class we need using the *from ... import* syntax:
- *from database import Database*
- *db = Database()*

Continue

- If, for some reason, products already has a class called Database , and we don't want the two names to be confused, we can rename the class when used inside the products module:
- *from database import Database as DB*
- *db = DB()*
- We can also import multiple items in one statement. If our database module also contains a Query class, we can import both classes using:
- *from database import Database, Query*
- Some sources say that we can even import all classes and functions from the database module using this syntax:
- *from database import **
- *But **Don't do this**.*it clogges your namespace

Organizing the modules

- As a project grows into a collection of more and more modules, we may find that we want to add another level of abstraction, some kind of nested hierarchy on our modules' levels.
- Files, however, can go in folders and so can modules. A package is a collection of modules in a folder.
- The name of the package is the name of the folder.
- All we need to do to tell Python that a folder is a package and place a (normally empty) file in the folder named __init__.py .
- If we forget this file, we won't be able to import modules from that folder.
- Let's put our modules inside an ecommerce package in our working folder, which will also contain a main.py to start the program.
- Let's additionally add another package in the ecommerce package for various payment options.

Absolute imports

- Absolute imports specify the complete path to the module, function, or path we want to import.
- If we need access to the Product class inside the products module, we could use any of these syntaxes to do an absolute import:
- *import ecommerce.products*
- *product = ecommerce.products.Product()*
- *or*
- *from ecommerce.products import Product*
- *product = Product()*
- *or*
- *from ecommerce import products*
- *product = products.Product()*
- These statements will work from any module using this syntax in main.py , in the database payment modules

```
parent_directory/
    main.py
    ecommerce/
            __init__.py
            database.py
            products.py
            payments/
                    __init__.py
                    paypal.py
                    authorizenet.py
```

Relative imports

- When working with related modules in a package, it seems kind of silly to specify the full path; we know what our parent module is named.
- This is where **relative imports** come in.
- Relative imports are basically a way of saying "find a class, function, or module as it is positioned relative to the current module".
- For example, if we are working in the products module and we want to import the Database class from the database module "next" to it, we could use a relative import:
- *from .database import Database*
- The period in front of database says, "Use the database module inside the current package"
- In this case, the current package is the package containing the products.py file we are currently editing, that is, the ecommerce package.
- If we were editing the paypal module inside the ecommerce.payments package, we would want to say, "Use the database package inside the parent package", instead.That is easily done with two periods:
- *from ..database import Database*

Creating Python classes

- The simplest class in Python 3 looks like this:
- class MyFirstClass:
- Pass
- The class definition starts with the class keyword. This is followed by a name (of our choice) identifying the class, and is terminated with a colon.
- We simply use the ***pass*** keyword on the second line to indicate that no further action needs to be taken.
- >>> a = MyFirstClass()
- >>> b = MyFirstClass()
- >>> print(a)
- <__main__.MyFirstClass object at 0xb7b7faec>
- This code instantiates two objects from the new class, named a and b
- When printed, the two objects tell us what class they are and what memory address they live at.

Adding attributes

- Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?
- class Point:
  - pass
- p1 = Point()
- p2 = Point()
- p1.x = 5
- p1.y = 4
- p2.x = 3
- p2.y = 6
- print(p1.x, p1.y)
- print(p2.x, p2.y)
- This code creates an empty Point class with no data or behaviors. Then it creates two instances of that class and assigns each of those instances x and y coordinates to identify a point in two dimensions.

Making it do something

- class Point:
- def reset(self):
- self.x = 0
- self.y = 0
- p = Point()
- p.reset()
- print(p.x, p.y)
- A method in Python is identical to defining a function. The one difference between methods and normal functions is that all methods have one required argument. This argument is conventionally named *self* ;There's nothing stopping you, however, from calling it *this*
- The self argument to a method is simply a reference to the object that the method is being invoked on
- Notice that when we call the p.reset() method, we do not have to pass the self argument into it. Python automatically takes care of this for us. It knows we're calling a method on the p object

Continue

- However, the method really is just a function that happens to be on a class. Instead of calling the method on the object, we could invoke the function on the class, explicitly passing our object as the self argument:
- p = Point()
- Point.reset(p)
- print(p.x, p.y)
- import math
- class Point:
- def move(self, x, y):
- self.x = x
- self.y = y
- def reset(self):
- self.move(0, 0)
- def calculate_distance(self, other_point):
- return math.sqrt((self.x - other_point.x)**2 + (self.y - other_point.y)**2)

Continue

- # how to use it:
- point1 = Point()
- point2 = Point()
- point1.reset()
- point2.move(5,0)
- print(point2.calculate_distance(point1))
- point1.move(3,4)
- print(point1.calculate_distance(point2))
- print(point1.calculate_distance(point1))

Initializing the object

- >>> point = Point()
- >>> point.x = 5
- >>> print(point.x)
- Most object-oriented programming languages have the concept of a constructor, a special method that creates and initializes the object when it is created.
- Python is a little different; it has a constructor and an initializer.
- Normally, the constructor function is rarely ever used unless you're doing something exotic
- The Python initialization method is the same as any other method, except it has a special name: __init__
- The leading and trailing double underscores mean, "this is a special method that the Python interpreter will treat as a special case"
- Let's start with an initialization function on our Point class that requires the user to supply x and y coordinates when the Point object is instantiated:

Continue

- class Point:
- def __init__(self, x, y):
- self.move(x, y)
- def move(self, x, y):
- self.x = x
- self.y = y
- def reset(self):
- self.move(0, 0)
- # Constructing a Point
- point = Point(3, 5)
- print(point.x, point.y)

Exception Handling

- The logic error leads to a runtime error that causes the program to terminate. These types of runtime errors are typically called exceptions.
- Programmers can create their own exceptions if they detect a situation in the program execution that warrants it.
- When an exception occurs, we say that it has been “raised.
- You can “handle” the exception that has been raised by using a try statement

Square root of a number

- *>>> anumber = int(input("Please enter an integer "))*
- *Please enter an integer -23*
- *>>> print(math.sqrt(anumber))*
- *We can handle this exception by calling the print function from within a try block. A corresponding except block “catches” the exception and prints a message back to the user in the event that an exception occurs.*
- *>>> try:*
- *print(math.sqrt(anumber))*
- *except:*
- *print("Bad Value for square root")*
- *print("Using absolute value instead")*
- *print(math.sqrt(abs(anumber)))*

Cont....

- It is also possible for a programmer to cause a runtime exception by using the raise statement.
- For example, instead of calling the square root function with a negative number, we could have checked the value first and then raised our own exception.
- NB: the program would still terminate but now the exception that caused the termination is something explicitly created by the programmer.
- *>>> if anumber < 0:*
- *... raise RuntimeError("You can't use a negative number")*
- *... else:*
- *... print(math.sqrt(anumber))*

What is a Stack?

- A stack (sometimes called a “push-down stack”) is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end.
- This end is commonly referred to as the “top.” The end opposite the top is known as the “base.”
- The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest.
- The most recently added item is the one that is in position to be removed first.
- This ordering principle is sometimes called LIFO, last-in first-out.
- Almost any cafeteria has a stack of trays or plates where you take the one at the top, uncovering a new tray or plate for the next customer in line

The Stack Abstract Data Type

- Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.
- push(item) adds a new item to the top of the stack. It needs the item and returns nothing.
- pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items on the stack. It needs no parameters and returns an integer.

if s is a stack that has been created and starts out empty,

Stack Operation	Stack Contents	Return Value
`s.isEmpty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.isEmpty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Stack Code

```
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

Continue

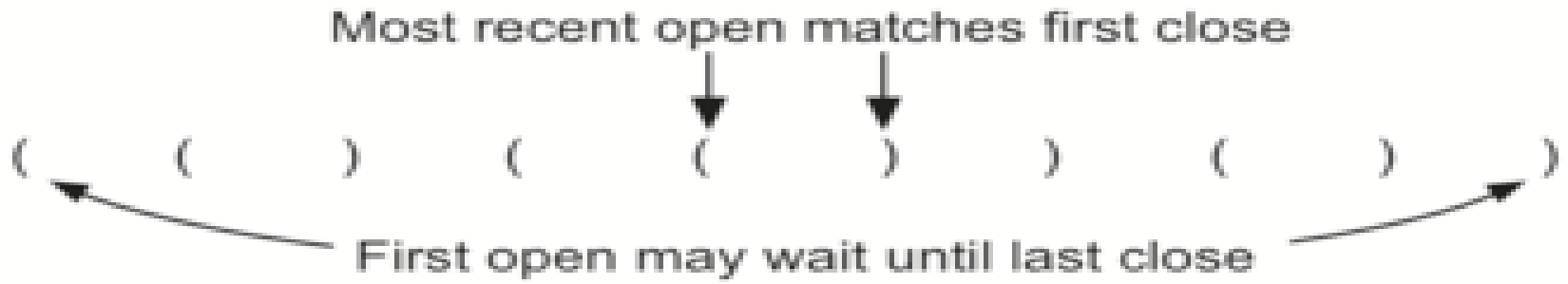
- Notice that the definition of the Stack class is imported from the pythonds module.
- from pythonds.basic.stack import Stack
- *s=Stack()*
- *print(s.isEmpty())*
- *s.push(4)*
- *s.push('dog')*
- *print(s.peek())*
- *s.push(True)*
- *print(s.size())*
- *print(s.isEmpty())*

Simple Balanced Parentheses

. Balanced parentheses means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested.

(()()()())
(((())))
(()((()))))
()))
(()()(()

The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.



Python Code

```
from pythonds.basic.stack import Stack
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False
print(parChecker('((()))'))
print(parChecker('(()'))
```