Essentials
(Some Advanced Topics)

IAB207 Semester 2 - 2019

```python
# User enters the year
year = int(input("Enter Year: "))

# Leap Year Check
if year % 4 == 0 and year % 100 != 0:
    print(year, "is a Leap Year")
elif year % 100 == 0:
    print(year, "is not a Leap Year")
elif year % 400 ==0:
    print(year, "is a Leap Year")
else:
    print(year, "is not a Leap Year")
```

```python
# definition of subclass starts here
class Student(Person):
    studentId = ""

    def __init__(self, student_name, student_age, student_id):
        super().__init__(student_name, student_age)
        self.studentId = student_id

    def get_id(self):
        return self.studentId  # returns the value of student
```

Aims of this lecture

- Python Language basics
  - Reminder of programming language concepts
- Object Oriented concepts
  - Definition of a class
  - Class relationships
- Organizing code in Python

Python set up

- Python3 will be used for the unit

- What is an Integrated Development Environment (IDE)?
  - Just makes development easy – **BIG** improvement in developer productivity
  - Code completion, code browser, syntax error highlighting
  - Debug support and variable watch

- There are many IDEs for Python
  - PyCharm – community edition and specifically for Python
  - Visual Studio Code – code editor from Microsoft
  - Atom – Code editor has support for Python
  - IDLE – Light weight Python IDE comes by default

Essentials of Python Language

- Variables and Types
- Data structures
- Branching
- Loops
- Functions

Syntax of the language

Scalar Types

- Types
  - **`int`** - Integers (whole numbers)
  - **`float`** – Real numbers (3.14, 2.6)
  - **`bool`** – True, False
  - **`NoneType`** – None (only one value)

- Function that return the type
  - **`type(5)`** – returns the type of 5 (int)

- Type conversion
  - **`int(3.78)`** result in 3
  - **`float(5)`** result in 5.0

Operators

Operator	Name	Example
+	Addition	X+Y
-	Subtraction	X-Y
*	Multiplication	X*Y
/	Division	X/Y
**	Exponentiation	X**2
%	Modulus	15%4 = ?

- Arithmetic operators on scalar types: `int` and `float`
- What happens if you use these operators on `bool`?

Variables – Store type values

- Has a name that is used to store and refer to the value

- Immutable
  - Once the value has been stored, it cannot be updated

```
x=5
p=3.0
q=True

val = type(5)
val = type(x)
```

Assignment, there is always just one variable name to left of assignment operator, where the value is **stored** (unlike math)

Strings and Operators

- Sequence of Characters
  - Characters could be letters, digits, spaces
  - Strings are immutable - once created they cannot be modified

```
x =‘Hello’
y=“World!”
```

- Concatenate string

```
z=x+y
```

- The * operator on string

```
x*2
```

List (Data structure – collection of Data)

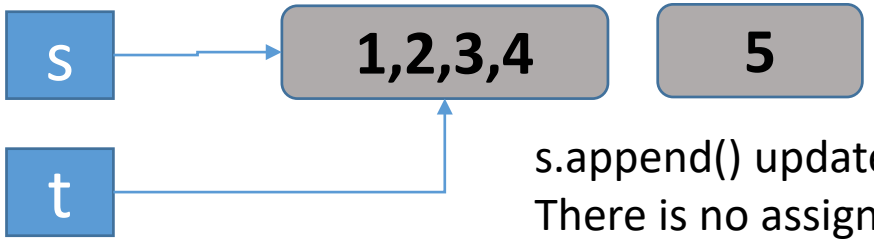
- List : sequences of elements (could have different element types)
- Lists are mutable – they can be modified and updated after their creation

```
room_types=[]

room_types.append('king')
room_types.append('queen')
room_types.append('standard')

print(room_types)
```

```
s=[1,2,3,4]
t=s
s.append(5)
print(t)
```



s.append() updates
There is no assignment of new value to s

List (contd..)

- operations on list
  - `sort()`
  - `remove()`
  - `pop()`

```
>>> room_list=['standard','king','queen']  1
>>> len(room_list)
3
>>> room_list[1]  2
'king'
>>> room_list.append('single')  3
>>> room_list
['standard', 'king', 'queen', 'single']
>>> room_list.pop()  4
'single'
>>> room_list
['standard', 'king', 'queen']
>>> room_list.insert(1,'single')  5
>>> room_list
['standard', 'single', 'king', 'queen']
```

List (contd..)

- Operations on list
  - count( )

```
>>> room_list.count()
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    room_list.count()
TypeError: count() takes exactly one argument (0 given)
>>> room_list.count('king')
1
>>> room_list.append('king')
>>> room_list
['standard', 'single', 'king', 'queen', 'king']
>>> room_list.count('king')
2
```

Dictionaries

- Need to keep a track of room_type and the price of the room

```
room_type =[‘standard’, ‘double’, ‘queens’]
room_price_list =[ 110,230,140]
```

- Separate list and each list should have the same length

```
idx= room_type.index(‘single’)
r_price = room_price_list[idx]
```

- Maintaining consistency when adding some information can be difficult

Dictionary (contd.)

- Store key value pairs

```
rooms={‘double’:230, ‘standard’:110 } #initialize with value
```

- Not ordered – can be accessed only using the keys

```
>>> rooms=dict()
>>> rooms['standard']=110
>>> rooms['double']=230
>>> print(rooms)
{'standard': 110, 'double': 230}
>>> rooms[1]
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    rooms[1]
KeyError: 1
>>> rooms['double']
230
```

Dictionary (contd.)

- Access functions

```
>>> rooms.keys()
dict_keys(['standard', 'double'])
>>> rooms.values()
dict_values([110, 230])
>>> rooms.items()
dict_items([('standard', 110), ('double', 230)])
```

- Check if a key exits

```
>>> 'standard' in rooms
True
>>> 'Standard' in rooms
False
```

Comparisons – integer, float and string

- <= less than or equal to
- >= greater than or equal to
- < less than
- > greater than
- == equality (different from assignment)

Comparisons return a bool value **`(True, False)`**

Operations on the Boolean value

- **not** : not (a>5)
- **and** : (a>5) and (b<=3)
- **or** : (a>5) or (b<=3)

```
a=10
b=20

print(a>10)

print(not a>10)

print(a>10 and b<100)

print(a>10 or b<100)
```

Branching

If condition:
    do something
else:
    do something different

```
if(a>10):
    print('a is high')
else:
    print('a is low')
```

If condition:
    do something
elif condition x:
    do something different
else:
    Nothing worked so do this

IFB104 example

```
speed=2

if speed<5:
    print('Hurry up')
elif speed<50:
    print('Go a bit faster')
elif speed<90:
    print('That is fast enough')
else:
    print('slow down')
```

It satisfies all the if and elif conditions
But only the code block with first condition that is satisfied is executed

Loops

- For loop
  - for a **variable** with values in some range
  - Run the block of code

```
for n in range(5): # n will take value 0,1,2,3,4
    print('current value loop',n)
```

```
for n in range(2,5): # n will take value 2,3,4
    print('current value loop ',n)
```

  - range(start, stop, step)
    - One argument – 0, stop,1
    - Two arguments – start, stop,1
  - Execute until stop-1

```
for n in range(2,8,2): # n will take value 2,4,6
    print('current value loop',n)
```

Loops

- **while** loop
  - Check for a condition
  - Run the block of code
  - Until the condition is true

```
n=0
while n<5:
    print('current value:', n)
    n=n+1
```

Breaking from loops

- Need to exit the loop before the condition
- Will exit the loop where the break statement is

```
for n in range(5): # n will take value 0,1,2,3,4
    print("Value of n:", n)
    for j in range(5):
        if(n==2 and j==2):
            print('Met the condition')
            break
        else:
            print("Value of j", j)
    print('In the outer loop')
```

Functions

- Logical portions of code as function
  - Abstraction – use it without knowing the internals
  - Decomposition – Do small parts of the big feature

- Write code that does a small task (decomposed task) and make it a black-box (abstract)

- What does the function contain?
  - Name
  - Input parameters
  - Return value

Functions

Function name

Defining a function here with **def** keyword

Input parameters (formal parameter)

Defining the parameter you are expecting –the type is not mentioned

```
total_rooms=40

def check_room_availability(date):
    booked_rooms = get_booked_rooms(date)
    if booked_rooms<total_rooms:
        return True
    else:
        return False
```

Return value

A function always returns a value
If you do not add return value
Python will add ***return None***

Function Signatures

- Default parameters

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closed=True, opener=None)

```
my_file = open('test.txt', 'w')
```

- Variable number of parameters

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

```
print('printing in python', 'hello', 'world')
```

- Variable number of **named** parameters

render_template(template_name, **context)

```
render_template('x.html', myarg1=5.3, myarg2='hello')
```

Object Oriented Programming

- Defining objects and their characteristics
- Objects are abstractions of common characteristics and behaviour



Person

- Data structure – List, Dictionary is an Object
  - Characteristic of List – store sequence of data
  - Behaviour or Operations – insert, delete, count

Objects

- Object represents the general characteristics and operations (behaviour)

- Data representing the Object
  - Person object - **`name, height, weight`**

- Operations to interact with the Object
  - **`increase_height(), decrease_weight(), increase_weight()`**

Class and Object

- Class is the Blueprint (our own custom type)
  - Data attributes
  - Operations/ Behaviours/ methods

- Object is the instance of the class
  - jill = Person(name, height, weight)

Class Definition

```
class Room:
    def __init__(self, name, id, price):
        self._name=name
        self._id=id
        ……….
```

Every method has self passed as first parameter

Helps identify particular instance of the class - myself

Special method to initialize the data attributes of the class

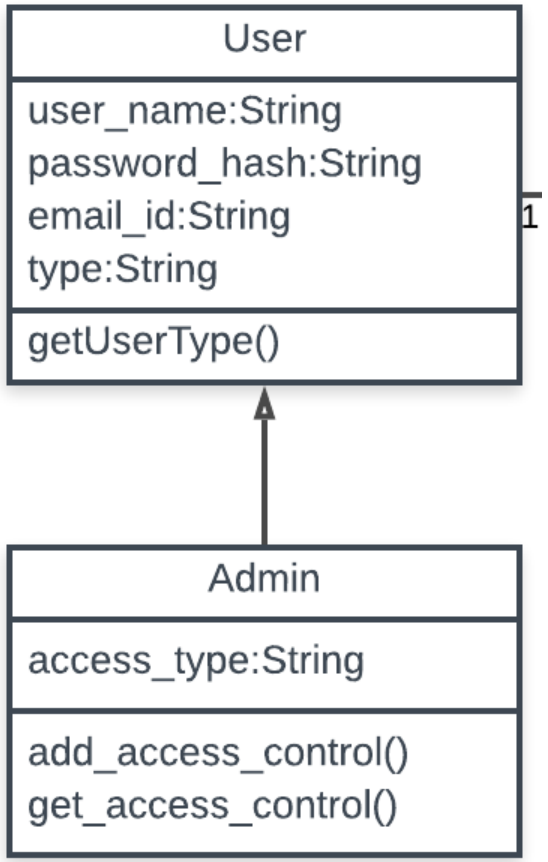
Class Definition

- `__init__`
- `__str__`
- `__repr__`
- Special methods that are not explicitly called programmers
- Python calls these in certain scenarios

```
def __str__(self):
    s="Name : " +self._name
    return s
```

Inheritance

- Define a User – person browsing the web site
- Need an **Admin** User who has additional privileges



Inheritance

- Reuse functionality available in **User**

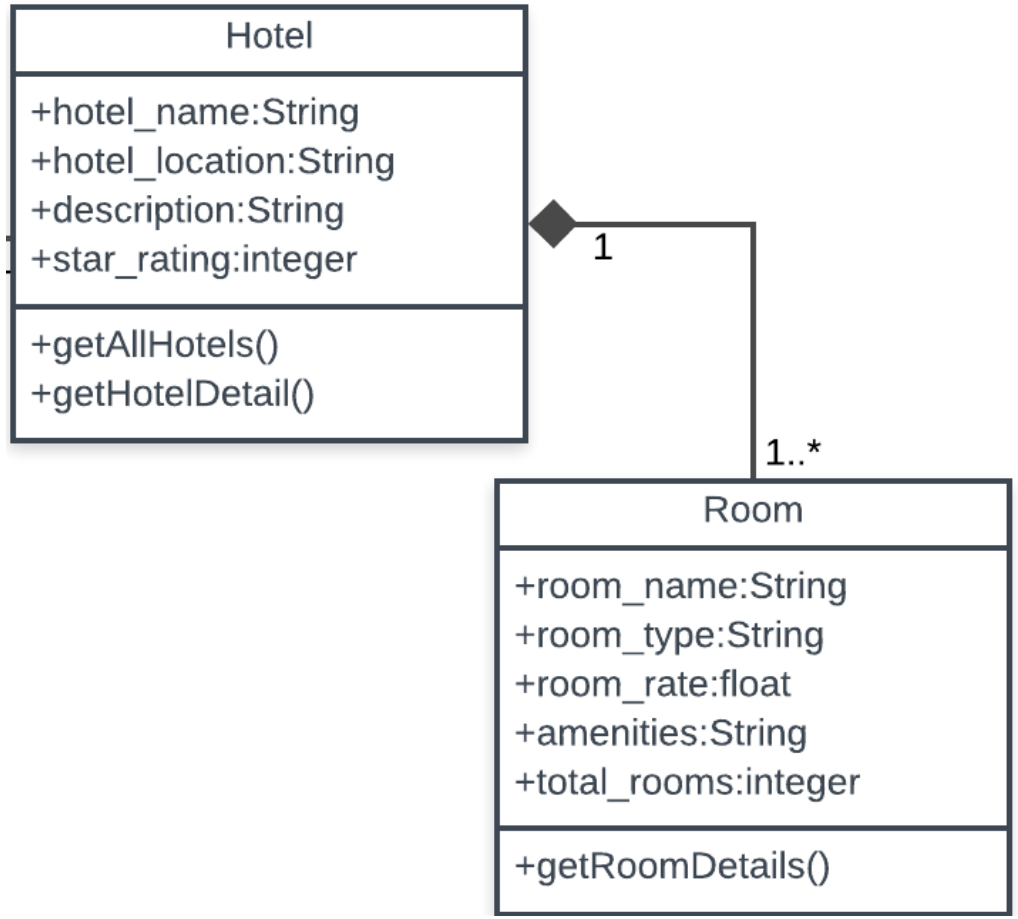
- In __init__ method of **Admin**
  - call User __init__ method
  - Write any additional code required for Admin.

```
class Admin(User):  # Admin is-a User
    def __init__(self, name, email)
        super().__init__(name,email)
        self.type='admin'
```

Aggregation

- Two types of relationships
  - Composition
  - Aggregation

- Difference in implementing the class relationships



Modules

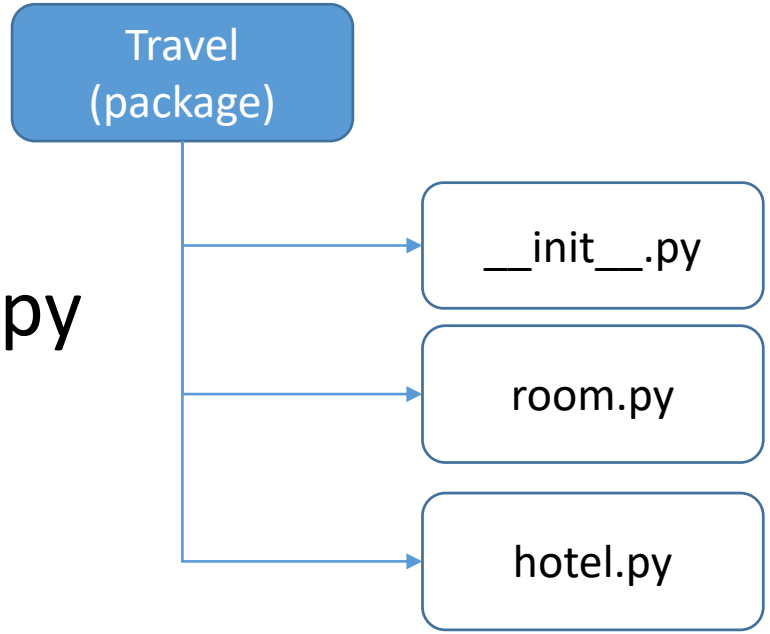
- Every python file is a module
- File room.py has the class definition Room
- Modules are used to define functions, classes

```
from room import Room
```

module

Packages

- Packages are collection of modules
- It is a directory containing all modules
- Package needs a special file called __init__.py
  - The file can be empty.




```
from Travel import hotel
x= hotel.Hotel()

from Travel.hotel import Hotel
x=Hotel()
```

Packages can have packages and modules

```
from travel.flights.booking import book_airline
```

```
travel/                                  # Top level package
    __init__.py                          # initialize travel package
    flights/                             # sub-package
        __init__.py
        booking.py
        frequent_flyer.py
    hotels/
    destinations/
```

Summary

- Python Language basics
  - Reminder of programming language concepts
- Object Oriented concepts
  - Definition of a class
  - Class relationships
- Organizing code in Python
  - Modules, Packages