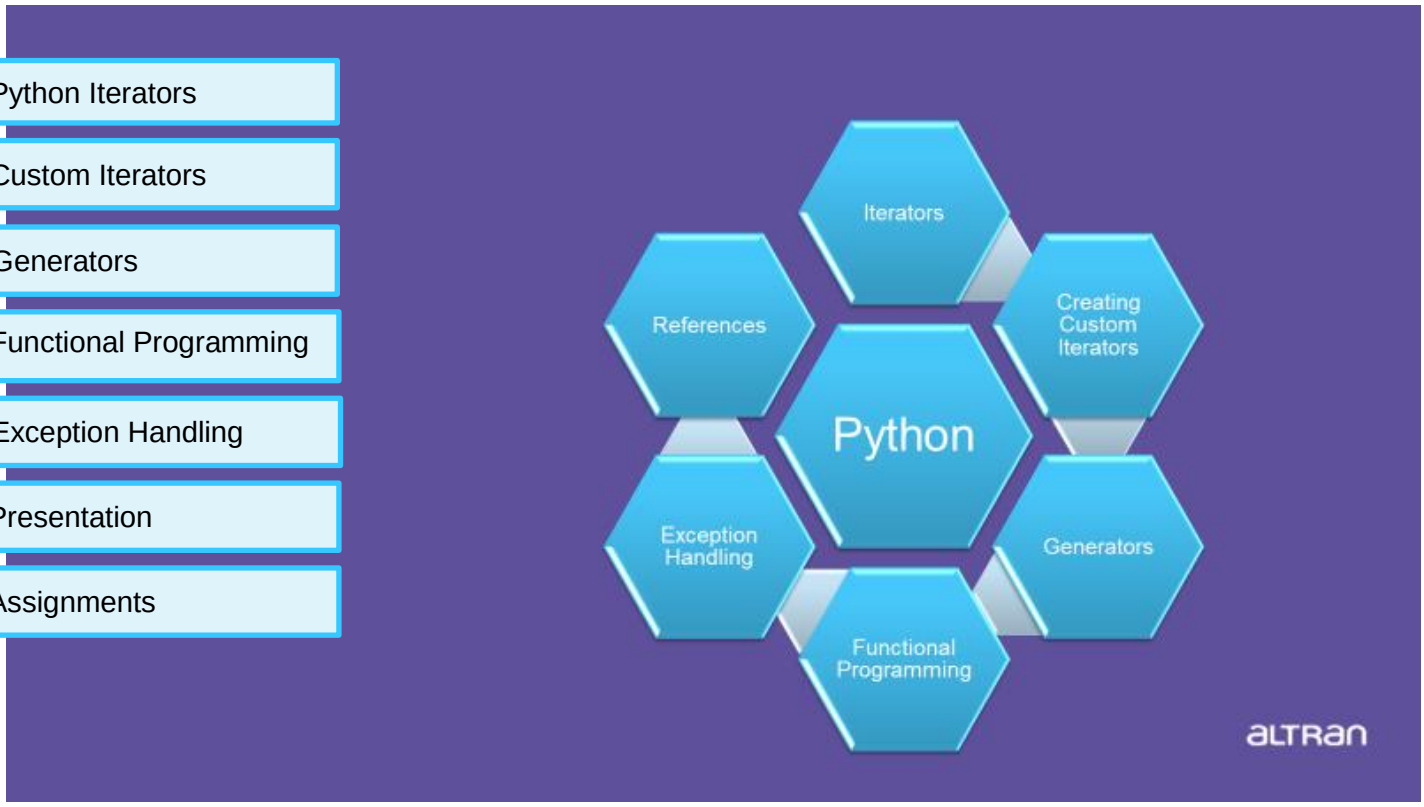

Python Iterators

An object whose class consists of an **"__iter__"** method which returns **"self"** and a **"__next__"** (next in Python 2) method is called an iterator. It is assumed that the class defines "__next__()", then __iter__() can just return self:

In general, the **"__iter__"** function returns an iterator object that defines the method "**__next__**" which accesses elements in the container one at a time.

When there are no more elements, "__next__()" raises a StopIteration exception which tells the for loop to terminate.

In simple words the variable that is used to loop over most of the "**iterables**" like container objects (like tuples, lists, sets, dicts, strings..) can be called an iterator.

To know more about Iterators: - 

(to refer documentation)

Conceptual Video on Iterators: - 

(to watch the video)

Creating Custom Iterators

Conceptual Video on Creating Custom Iterators: - 

(to watch the video)

```
"""Demo for iterator...Generates 0, 1, 2, ...(max_range - 1)"""
class DemoIterator:
    """Class having a __iter__ & __next__ method"""
    def __init__(self, n):
        self.cur_val = 0
        self.max_range = n
    def __iter__(self):
        """Should return a object having __next__method"""
        return self
    def __next__(self):
        if self.cur_val < self.max_range:
            val = self.cur_val
            self.cur_val += 1
            return val
        else:
            raise StopIteration()
def main():
    """main function"""
    mobj = DemoIterator(5)
    #Internally StopIteration terminates the for loop
    for myitr in myobj:
        print(myitr)
if __name__ == "__main__":
    main()
```

Output
0
1
2
3
4

Key-Points to Note:

- myobj object is instantiated from DemoIterator Class.
- The class is having methods “__iter__” and “__next__”
- “__iter__” returns self (Since the “__next__” is defined by this class.
- “__next__()” raises a StopIteration exception which tells the for loop to terminate.
- The raise statement allows the programmer to force a specified exception to occur.

Generators

- Generators are a simple and powerful tool for creating iterators.
- Generators are written like regular functions
- The yield statement is used to return data from generators
- Each time next() is called on it , the generator resumes where it left off (it remembers all the data values and which statement was last executed). If a “for” loop is used, “next()” is called internally.

Why Generators are useful?

- Generators provide an easy, built-in way to create instances of Iterators.
- A function with yield in it is still a function, that, when called, returns an instance of a generator object:
- Anything that can be done with generators can also be done with class-based iterators
- A Generators is compact since the __iter__() and __next__() methods are created automatically.
- “StopIteration” is also raised automatically
- Considering these, we can say that it is easy to create iterators with Generators.

To know more about Generators: - 

(to refer documentation)

Conceptual Video on Generators: - 


```
"""Demo for Generator Generates 0, 1, 2, ...(max_range - 1)"""
def DemoIterator(n):
    """Class having a __iter__ & __next__ method"""
    cur_val = 0
    max_range = n
    while (cur_val < max_range):
        yield cur_val
        cur_val += 1

def main():
    """main function"""
    myitr = DemoIterator(5)
    for var in myitr:
        print(var)

if __name__ == "__main__":
    main()
```

```
"""Demo for Generator Generates 0, 1, 2, ...(max_range - 1)"""
def DemoIterator(n):
    """Class having a __iter__ & __next__ method"""
    cur_val = 0
    max_range = n
    while (cur_val < max_range):
        yield cur_val
        cur_val += 1

def main():
    """main function"""
    myitr = DemoIterator(5)
    print(next(myitr))
    print(next(myitr))
    print(next(myitr))
    print(next(myitr))
    print(next(myitr))
    print(next(myitr))

if __name__ == "__main__":
    main()
```

Key-Points to Note:

- In one code, for statement is used to get values from the iterator (created by a generator function)
- In the other code, explicit “next” calls are made to get the values.
- The last “next” call (that is the sixth one) will raise a “StopIteration” automatically.

Functional Programming

In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements.

In a functional program, input flows through a set of functions. Each function operates on its input and produces some output. Functional style discourages functions with side effects that modify internal state or make other changes that aren’t visible in the function’s return value.

Example of Functional Programming: -

Problem Statement:
Get an input list of numbers and generate a new list of the numbers squared.

(An example of Procedural Style)

```
def square(x):
    return x*x
vals = [1, 2, 3, 4]
newvals = []
for v in vals:
    newvals.append(square(v))
```

(An example of Functional Style)

```
vals = [1, 2, 3, 4]
newvals = map(lambda x:x*x, vals)
```

- Shorter Code
- Use of Anonymous function(lambda)
- “map” applies a function to each of the elements of a list and creates a new iterator

To know more about Functional Programming: - 

and 

Functional Programming Features: -

Generator expressions and list comprehensions	Click Here
Built in functions like map(), filter()	Click Here & Click Here
Small functions and the lambda expression	Click Here
First Class Objects	Click Here

Advantages of Functional Programming: -

- It is easier to construct a mathematical proof that a functional program is correct.
- Modularity
- Ease of debugging and testing

Exception Handling

What is a Syntax Error?

A Syntax error is an error in the syntax of a sequence of characters (or tokens) that is intended to be written in a programming language.

What is an Exception?

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions. Exceptions can be handled.

Syntax Error

```
for i in [1, 2, 3, 4, 5]
            print(i)
File "main.py", line 1
  for i in [1, 2, 3, 4, 5]
               ^
SyntaxError: invalid syntax
```

Exception

```
mylist =  ["Melhi", "Mumbai", "Chennai", "Bangalore"]
print(sum(mylist))

$python3 main.py
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print(sum(mylist))
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

More about Exception Handling: -

Handling Exceptions	**Click Here**
Clean-up Actions	**Click Here**
Built-in Exceptions	**Click Here**
File Not Found Error	**Click Here**

Presentation

Assignments