lambda expression

Syntax

```
lambda arguments : expression
```

ex1

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))         # 15
```

ex2

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))         #30
```

Errors, Exceptions & File I/O In Python

Syntax errors

Syntax errors occur when the Python interpreter encounters a line of code that it cannot interpret. These errors are usually caused by a typo or a missing character. For example, if you forget to close a parenthesis, the interpreter will raise a syntax error.

Exceptions

Exceptions, on the other hand, occur when the program encounters an abnormal condition that it cannot handle. These can include things like attempting to divide by zero, trying to access an element in a list that does not exist, or trying to open a file that does not exist. Exceptions are raised using the `raise` statement. When an exception is raised, the normal flow of execution is interrupted and the program jumps to the nearest exception handler. If no exception handler is found, the program will terminate.

Python provides several built-in exceptions that can be used to handle these types of errors. Some common examples include:

- `NameError`: raised when a variable is not defined
- `TypeError`: raised when an operation is attempted on a value of an inappropriate type
- `IndexError`: raised when a list index is out of range
- `KeyError`: raised when a dictionary key is not found
- `IOError`: raised when an input/output operation fails
- `ZeroDivisionError`: raised when a number is divided by zero

To handle exceptions, you can use a try-except block. The code that may raise an exception is placed in the try block, and the code to handle the exception is placed in the except block. For example:

```
try:
    x = 1/0
except ZeroDivisionError:
    print("You can't divide by zero!")
```

You can also catch multiple exceptions in the same block using parentheses.

```
try:
    x = 1/0
except (ZeroDivisionError, TypeError):
    print("You can't divide by zero or operation not support on given 
type!")
```

It is also possible to use an else block to specify code that should be executed if the try block completes without raising an exception. Finally block is used to specify code that should always be executed, regardless of whether an exception was raised or not.

```
try:
    x = 1/0
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print("No exception was raised.")
finally:
    print("This will always be executed.")
```

In addition to the built-in exceptions, you can also create your own custom exceptions by creating a new class that inherits from the `Exception` class.
Example:

```
class InvalidAgeError(Exception):
    def __init__(self, message):
        self.message = message

age = -5
if age < 0:
    raise InvalidAgeError("Age cannot be negative.")
```

In this example, we define a new class called `InvalidAgeError` that inherits from the `Exception` class. The `__init__` method takes a message as an argument, which can be used to provide more information about the error.
In the main program, we check if the value of `age` is less than 0. If it is, we raise an instance of the `InvalidAgeError` exception, passing in a message that indicates that the age cannot be negative.
You can catch this custom exception in the same way as built-in exceptions:

```
try:
    if age < 0:
        raise InvalidAgeError("Age cannot be negative.")
except InvalidAgeError as e:
    print(e)
```

It is also possible to use the custom exception in conjunction with the built-in exceptions, by catching it in the same block.

```
try:
    if age < 0:
        raise InvalidAgeError("Age cannot be negative.")
    x = 1/0
except (InvalidAgeError, ZeroDivisionError) as e:
    print(e)
```

In this way, you can create custom exceptions to handle specific error conditions that are not covered by the built-in exceptions, making your code more readable and manageable.

File Input/Output

File Input/Output (I/O) is a fundamental aspect of programming and is used to read data from files and write data to files. In Python, file I/O is performed using built-in functions and classes such as `open()`, `file()`, `read()`, `write()`, and `close()`.

Opening a File

The first step in working with a file is opening it. The `open()` function is used to open a file. The function takes two arguments: the file name and the mode in which the file should be opened.

```
file = open("file.txt", "r")
```

The first argument is the name of the file, and the second argument is the mode in which the file should be opened. The most commonly used modes are “r” (read), “w” (write), and “a” (append). If the file does not exist and you open it in “w” mode, a new file will be created. If the file does exist, it will be truncated and overwritten. If the file does not exist and you open it in “a” mode, a new file will be created. If the file does exist, the data will be appended to the end of the file.

Reading a File

Once a file is opened, you can read its contents. The `read()` method is used to read the entire contents of a file.

```
file_contents = file.read()
```

You can also read a specific number of characters from a file using the `read(n)` method, where n is the number of characters to be read.

```
file_contents = file.read(5)
```

You can also read a file line by line using the `readline()` method.

```
file_contents = file.readline()
```

Writing to a File

You can write to a file using the `write()` method.

```
file = open("file.txt", "w")
file.write("Hello World!")
```

You can also use the `writelines()` method to write a list of strings to a file.

```
lines = ["Hello World!", "This is a test."]
file.writelines(lines)
```

Closing a File

It is important to close a file after you have finished working with it.
The `close()` method is used to close a file.

```
file.close()
```

Binary Files

In addition to text files, Python can also work with binary files. Binary files are used to store data in a format that is not human-readable. To work with binary files, you can use the `open()` function with the “b” mode.

```
binary_file = open("file.bin", "rb")
```

The methods for reading and writing to binary files are the same as for text files, but the data that is read or written will be in binary format.

with open

It’s also worth noting that python 3.x has `with open` statement that can be used to open the file, you don’t have to explicitly close the file as it automatically closes the file once the block of code is done executing.

```
with open("file.txt", "r") as file:
    file_contents = file.read()
    # Do something with file_contents
```

In this way, you don’t have to worry about closing the file, if an exception is raised or the program exits unexpectedly, the file will still be closed properly.

Another important point to keep in mind is handling exceptions while working with files. If a file does not exist or the file path is incorrect, the `open()` function will raise a `FileNotFoundError` exception. Similarly, if you try to read from or write to a file that is not open, a `ValueError` will be raised.
Therefore, it is important to always use try-except blocks when working with files to handle any potential errors that may occur.

```
file_opened = False
try:
    file = open("file.txt", "r")
    file_opened = True
    file_contents = file.read()
except FileNotFoundError:
    print("The file was not found.")
except ValueError:
    print("The file is not open.")
finally:
    if file_opened:
        file.close()
```

In this example, the try block attempts to open and read the contents of the file “file.txt”. If the file is not found, the program will execute the code in the except FileNotFoundError block and print an error message. If the file is not open, the program will execute the code in the except ValueError block and print an error message. In both cases, the code in the finally block will execute, ensuring that the file is closed properly if it was opened.