Cheat Sheet

Errors & Exceptions

There are two major kinds of errors:

- 1. Syntax Errors
- 2. Exceptions

Syntax Errors

Syntax errors are parsing errors which occur when the code is not adhering to **Python Syntax**.

Code

PYTHON

```
1 if True print("Hello")
```

Output

```
SyntaxError: invalid syntax
```

When there is a syntax error, the program will **not** execute even if that part of code is not used.

Code

PYTHON

```
1 print("Hello")
2
3 def greet():
4 print("World"
```

Output

SyntaxError: unexpected EOF while parsing

Notice that in the above code, the syntax error is inside the

greet function, which is not used in rest of the code.

Exceptions

Even when 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**.

Example Scenario

We wrote a program to download a Video over the Internet.

- Internet is disconnected during the download
- We do not have space left on the device to download the video

Example 1

Division Example

Input given by the user is not within expected values.

Code

PYTHON

```
1 def divide(a, b):
2     return a / b
3
4 divide(5, 0)
```

Output

ZeroDivisionError: division by zero

Example 2

Input given by the user is not within expected values.

Code

PYTHON

```
1
2 def divide(a, b):
3    return a / b
4
5 divide("5", "10")
```

Output

```
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Example 3

Consider the following code, which is used to update the quantity of items in store.

Code

```
1 class Store:
2    def __init__(self):
3        self.items = {
4         "milk" : 20, "bread" : 30, }
5
6    def add_item(self, name, quantity):
7        self.items[name] += quantity
8
9    s = Store()
10    s.add_item()'biscuits', 10)
```

Output

KeyError: 'biscuits'

Working With Exceptions

What happens when your code runs into an exception during execution?

The application/program crashes.

End-User Applications

When you develop applications that are directly used by end-users, you need to **handle different possible exceptions** in your code so that the application will not crash.

Reusable Modules

When you develop modules that are used by other developers, you should **raise exceptions** for different scenarios so that other developers can handle them.

Money Transfer App Scenario

Let's consider we are creating an app that allows users to transfer money between them.



Develop a Class to model Bank Account Use Bank Account class to implement Money Transfer





Bank Account Class

Example 1

| User 1 | User 2 | Balance | 250/- | Balance | 100/-

transfer_amount(user_1, user_2, 50)

Balance 200/-

Balance 150/-





Code

```
class BankAccount:
def __init__(self, account_number):
self.account_number = str(account_number)
self.balance = 0

def get_balance(self):
return self.balance
```

```
ŏ
9
        def withdraw(self, amount):
            if self.balance >= amount:
10
                self.balance -= amount
11
12
            else:
13
                print("Insufficient Funds")
14
15
        def deposit(self, amount):
            self.balance += amount
16
17
18
    def transfer_amount(acc_1, acc_2, amount):
19
20
        acc 1.withdraw(amount)
        acc_2.deposit(amount)
21
22
23
24 user 1 = BankAccount("001")
25 user_2 = BankAccount("002")
26 user_1.deposit(250)
27  user 2.deposit(100)
28
29 print("User 1 Balance: {}/-".format(user_1.get_balance()))
30 print("User 2 Balance: {}/-".format(user_2.get_balance()))
31 transfer_amount(user_1, user_2, 50)
32 print("Transferring 50/- from User 1 to User 2")
   print("User 1 Balance: {}/-".format(user_1.get_balance()))
    print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ^

Output

```
User 1 Balance: 250/-
User 2 Balance: 100/-
Transferring 50/- from User 1 to User 2
User 1 Balance: 200/-
User 2 Balance: 150/-
```

Example 2





```
class BankAccount:
        def __init__(self, account_number):
            self.account_number = str(account_number)
 3
 4
            self.balance = 0
 5
        def get balance(self):
 6
 7
            return self.balance
 8
 9
        def withdraw(self, amount):
            if self.balance >= amount:
10
                self.balance -= amount
11
12
            else:
                print("Insufficient Funds")
13
14
        def deposit(self, amount):
15
            self.balance += amount
16
17
18
19
    def transfer_amount(acc_1, acc_2, amount):
        acc_1.withdraw(amount)
20
        acc 2.deposit(amount)
21
22
23
   user 1 = BankAccount("001")
25
    user_2 = BankAccount("002")
    user 1.deposit(25)
26
27
    user 2.deposit(100)
28
    nrint("User 1 Balance: {}/-".format(user 1.get halance()))
```

```
print("User 2 Balance: {}/-".format(user_1.get_balance()))

transfer_amount(user_1, user_2, 50)

print("Transferring 50/- from User 1 to User 2")

print("User 1 Balance: {}/-".format(user_1.get_balance()))

print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ^

Output

User 1 Balance: 25/User 2 Balance: 100/Insufficient Funds
Transferring 50/- from User 1 to User 2
User 1 Balance: 25/User 2 Balance: 150/-

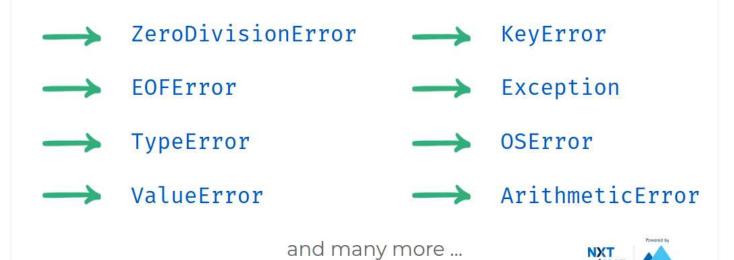
Raising Exceptions

When your code enters an unexpected state, raise an exception to communicate it.



Built-in Exceptions

Different **exception classes** which are raised in different scenarios.



You can use the built-in exception classes with **raise** keyword to **raise an exception** in the program.

Code

We can pass message as **argument**.

PYTHON

1 raise ValueError[]"Unexpected Value!!"[)

Output

ValueError:Unexpected Value!!

Bank Account Class

Example 1

User 1

Balance

25/-

User 2

Balance

100/-

```
transfer_amount(user_1, user_2, 50)
```

ValueError: Insufficient Funds







Code

```
class BankAccount:
 2
        def __init__(self, account_number):
            self.account_number = str(account_number)
 3
            self.balance = 0
 4
 5
 6
        def get_balance(self):
 7
            return self.balance
 8
        def withdraw(self, amount):
 9
            if self.balance >= amount:
10
                self.balance -= amount
11
12
            else:
                raise ValueError("Insufficient Funds")
13
14
        def deposit(self, amount):
15
            self.balance += amount
16
17
18
    def transfer_amount(acc_1, acc_2, amount):
19
        acc_1.withdraw(amount)
20
21
        acc_2.deposit(amount)
22
23
   user_1 = BankAccount("001")
24
   user 2 = BankAccount("002")
   user_1.deposit(25)
    user_2.deposit(100)
27
```

```
print("User 1 Balance: {}/-".format(user_1.get_balance()))
print("User 2 Balance: {}/-".format(user_2.get_balance()))
transfer_amount(user_1, user_2, 50)
print("Transferring 50/- from User 1 to User 2")
print("User 1 Balance: {}/-".format(user_1.get_balance()))
print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Output

```
User 1 Balance: 25/-
User 2 Balance: 100/-
```

ValueError: Insufficient Funds

Handling Exceptions

Python provides a way to **catch** the exceptions that were raised so that they can be properly handled.

- Exceptions can be handled with **try-except** block.
- Whenever an exception occurs at some line in try block, the execution stops at that line and jumps to except block.

PYTHON

Collapse ^

```
1 try:
2  # Write code that
3  # might cause exceptions.
4 except:
5  # The code to be run when
6  # there is an exception.
```

Transfer Amount

Example 1



Code

29

30

user_2 = BankAccount("002")

user_1.deposit(25)

PYTHON class BankAccount: def __init__(self, account_number): 3 self.account_number = str(account_number) self.balance = 0 4 5 def get_balance(self): 6 return self.balance 7 8 9 def withdraw(self, amount): if self.balance >= amount: 10 self.balance -= amount 11 12 else: raise ValueError("Insufficient Funds") 13 14 def deposit(self, amount): 15 self.balance += amount 16 17 18 19 def transfer amount(acc 1, acc 2, amount): try: 20 acc_1.withdraw(amount) 21 acc 2.deposit(amount) 22 23 return True 24 except: 25 return False 26 27 28 user 1 = BankAccount("001")

```
user_2.deposit(100)

print("User 1 Balance: {}/-".format(user_1.get_balance()))

print("User 2 Balance: {}/-".format(user_2.get_balance()))

print(transfer_amount(user_1, user_2, 50))

print("Transferring 50/- from User 1 to User 2")

print("User 1 Balance: {}/-".format(user_1.get_balance()))

print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Output

```
User 1 Balance: 25/-
User 2 Balance: 100/-
False
Transferring 50/- from User 1 to User 2
User 1 Balance: 25/-
User 2 Balance: 100/-
```

Summary

Reusable Modules

• While developing reusable modules, we need to raise Exceptions to stop our code from being used in a bad way.

End-User Applications

• While developing end-user applications, we need to handle Exceptions so that application will not crash when used.

Handling Specific Exceptions

We can specifically mention the **name of exception** to catch all exceptions of that specific type.

Syntax

```
1 try:
2  # Write code that
3  # might cause exceptions.
4 except Exception:
```

```
# The code to be run when # there is an exception.
```

Example 1

Code

PYTHON

```
1 try:
2    a = int(input())
3    b = int(input())
4    c = a/b
5    print(c)
6    except ZeroDivisionError:
7    print("Denominator can't be 0")
8    except:
9    print("Unhandled Exception")
```

Input

5

0

Output

Denominator can't be 0

Example 2

Code

Input given by the user is not within expected values.

```
1 try:
2     a = int(input())
3     b = int(input())
4     c = a/b
5     print(c)
6 except ZeroDivisionError:
```

```
7  print("Denominator can't be 0")
8  except:
9  print("Unhandled Exception")
```

Input

12

а

Output

Unhandled Exception

We can also access the handled exception in an **object**.

Syntax

PYTHON

```
1 try:
2  # Write code that
3  # might cause exceptions.
4 except Exception as e:
5  # The code to be run when
6  # there is an exception.
```

Code

```
class BankAccount:
        def __init__(self, account_number):
2
3
            self.account_number = str(account_number)
            self.balance = 0
4
5
        def get balance(self):
 6
            return self.balance
7
8
9
        def withdraw(self, amount):
            if self.balance >= amount:
10
11
                self.balance -= amount
```

```
12
            else:
13
                raise ValueError("Insufficient Funds")
14
15
        def deposit(self, amount):
            self.balance += amount
16
17
18
19
    def transfer_amount(acc_1, acc_2, amount):
20
        try:
            acc 1.withdraw(amount)
21
            acc 2.deposit(amount)
22
23
            return True
24
        except ValueError as e:
25
            print(str(e))
26
            print(type(e))
27
            print(e.args)
28
            return False
29
30 user 1 = BankAccount("001")
31 user 2 = BankAccount("002")
32 user_1.deposit(25)
33  user 2.deposit(100)
34
35 print("User 1 Balance: {}/-".format(user_1.get_balance()))
36 print("User 2 Balance: {}/-".format(user_2.get_balance()))
37 print(transfer amount(user 1, user 2, 50))
   print("Transferring 50/- from User 1 to User 2")
    print("User 1 Balance: {}/-".format(user 1.get balance()))
    print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ^

Output

```
User 1 Balance: 25/-
User 2 Balance: 100/-
Insufficient Funds
<class 'ValueError'>
('Insufficient Funds',)
False
Transferring 50/- from User 1 to User 2
User 1 Balance: 25/-
User 2 Balance: 100/-
```

Handling Multiple Exceptions

We can write **multiple exception blocks** to handle different types of exceptions differently.

Syntax

PYTHON

```
1 try:
2  # Write code that
3  # might cause exceptions.
4 except Exception1:
5  # The code to be run when
6  # there is an exception.
7 except Exception2:
8  # The code to be run when
9  # there is an exception.
```

Example 1

Code

PYTHON

```
1 try:
      a = int(input())
        b = int(input())
 4
        c = a/b
 5
        print(c)
 6
  except ZeroDivisionError:
 7
        print("Denominator can't be 0")
 8
   except ValueError:
        print("Input should be an integer")
 9
10
    except:
        print()"Something went wrong"()
11
```

Collapse ^

Input

5

0

Output

Denominator can't be 0

Example 2

Code

```
1 try:
    a = int(input())
      b = int(input())
 4
       c = a/b
 5
        print(c)
 6 except ZeroDivisionError:
 7
        print("Denominator can't be 0")
  except ValueError:
8
 9
        print("Input should be an integer")
    except:
10
        print()"Something went wrong"()
```

Input

12

а

Output

Input should be an integer

Submit Feedback

PYTHON

Collapse ^