Exceptions in coding

An exception is an unexpected error that occurs somewhere along the program as it is executed. An exception can cause the program to crash or unexpected results.

There are many things that can cause a program to throw an exception. Amongst these causes are:

- Out of range values
- Invalid use input
- Division by zero
- File not found
- And many more…

These exceptions can however be predicted and utilized. With the help of Exception handling techniques, we can write code that wraps around a block of code by using “try” and “except”.

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("Error: cannot divide by zero!")
```

In this example, instead of the program crashing and throwing an exception error, in this case ZeroDivisionError, the except block catches the exception and prints the message below instead.

Reference:

https://www.programiz.com/python-programming/exceptions

Classes in Python

Python is an object-oriented programming language (OOP) that often utilizes Classes to construct Objects. You can think of these classes as blueprints for creating new objects, called instances. In the class, which the object instance derives from, you can define characteristics and behavior for the object.

Classes are often used to replicate real-life objects and concepts, such as an animal or a person. For example:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def greet(self):
    print(f"Hello, my name is {self.name} and I'm {self.age} years old.")

person = Person("John", 30)
person.greet()  # Output: Hello, my name is John and I'm 30 years old.
```

In addition, classes maintain state. This means that a class can store data which persists over multiple function calls. This is very useful for storing data.

Objects in Python

Objects in python are instances of classes. This means that they are separate copies of the class. Each of these objects can store data as well as contain behavior based on the class, but not specific to it.

Classes can in turn take arguments to create these objects. An example of such an object is a playing card from a deck. Each card contains a rank and suit. Therefore, instead of creating 52 unique cards manually, you could create a class containing attributes that can save both suit and rank. Then you can generate a deck using just the one class/"blueprint". Example of a card class:

```
Codeium: Refactor | Explain
class card:
    Codeium: Refactor | Explain | Generate Docstring | ×
    def __init__(self, rank, suit):
        self.suit = suit
        self.rank = rank

    # Using __str__ to represent the string values of the card objects
    Codeium: Refactor | Explain | Generate Docstring | ×
    def __str__(self):
        return f"{self.rank} of {self.suit}"
```

Example of how to utilize the card class to create card objects using a function:

```
def make_deck():
    # Define the suits and ranks
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

    # Generate the deck using list comprehension
    deck = [cardCons.card(rank, suit) for suit in suits for rank in ranks]
    return deck
```

As you can see here, I have used list comprehension together with nested for loops to generate instances of cards from the cardCons class.