COSC 2306
Data Programming

OOP and Class

Generators

In-class programming:

Write Python codes to generate odd numbers up to a given limit **n** using the following methods:

1. An iterator class **OddNumberIterator** that using iterator.

2. A generator function **odd_number_generator** that yields odd numbers.

Generators

```
class OddNumberIterator:
    def __init__(self, n):
        self.n = n
        self.current = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.n:
            raise StopIteration
        odd = self.current
        self.current += 2
        return odd
```

```
def odd_number_generator(n):
    current = 1
    while current < n:
        yield current
        current += 2
```

Exception Handling

- Exceptions occur when certain *exceptional* situations occur in our program
- For example, what if we are reading a file and we accidentally deleted it in another window or some other error occurred? Such situations are handled using exceptions
- What if our program had some invalid statements?
- This is handled by Python which raises its hands and tells you there is an error

Exception Handling

- Consider a simple print statement

  >>> print 'Hello, World'

  Hello, World

- What if we misspelt print as Print (note the capitalization)?

  Python raises a syntax error

  >>> Print 'Hello, World’

  File "<stdin>", line 1 Print 'Hello, World' ^ SyntaxError: invalid syntax

- Observe that a SyntaxError is raised and also the location where the error was detected, is printed

  -- this is what a handler for the error does

Exception Handling

- To show the usage of exceptions, we will **try** to read integer input from the user and see what happens

```
a_number = int(input("Please enter an integer "))
print(math.sqrt(a_number))
>> Please enter an integer -23
Traceback (most recent call last):
File "<pyshell#102>", line 1, in <module>
print(math.sqrt(a_number))
ValueError: math domain error
```

- Python raises an error called ValueError: math domain error
- How to handle errors in a more user-friendly manner?

Exception Handling

- We can handle exceptions using the try...except statement
- We put all the statements that might raise an error in the try-block
- And we put all the error handlers in the except-block

Exception Handling

```
import math
a_number = int(input("Please enter an integer "))
try:
    print(math.sqrt(a_number))
except:
    print("Bad Value for square root")
    print("Using absolute value instead")
    print(math.sqrt(abs(a_number)))
```

```
>> Please enter an integer -23
Bad Value for square root
Using absolute value instead
4.795831523312719
```

Exception Handling

- The except-clause can handle a single specified error/exception or a parenthesized list of errors/exceptions
- For bare except (except:, i.e., no names of errors/exceptions are supplied), it will handle all errors/exceptions

  - This is not recommended as it may trap manual interrupt and thus hide bugs or suppress signals unintendedly

```
try:
    while True:
        pass
except:
    print("Caught an exception")
```

User's manual interrupt (KeyboardInterrupt) can be caught by the except, and the program does not stop as expected. This can confuse users trying to terminate the program

```
try:
    result = 10 / 0
except:
    print("An error occurred")
```

Exception is trapped w/o showing the error detail - hides the ZeroDivisionError and prevents the developer from knowing what actually went wrong, making debugging harder

Exception Handling

- Better to use except Exception instead to catches most runtime errors and exceptions (excludes system-exiting exceptions like SystemExit, KeyboardInterrupt, GeneratorExit)
- except Exception as x binds the exception instance to a variable (e.g., x), so its message or attributes can be accessed in the handler

```
try:
    result = 10 / 0
except Exception as x:
    print(f"An error occurred: {x}")   # An error occurred: division by zero

try:
    num = int("abc")  # This raises a ValueError
except Exception as x:
    print(f"Exception type: {type(x).__name__}")
    print(f"Exception args: {x.args}")
    print(f"Exception message: {x}")
```

```
Exception type: ValueError
Exception args: ("invalid literal for int() with base 10: 'abc'",)
Exception message: invalid literal for int() with base 10: 'abc'
```

Exception Handling

- At least one except clause for every try clause
- If any error or exception is not handled (or no try-except block), the default Python handler is called which stops the program execution and prints a message
- We can have an else clause with the try...catch block, which executes if no exception occurs

```
try:
    file=open('my_file')
except FileNotFoundError:
    print('This file is not exist.')
except Exception:
    print('Sorry, something went wrong.')
else:
    print(file.read())
    f.close()
```