Python Error Handling:
Exceptions

Python Error Handling

There are two possible types of error when developing programs

i. Syntax error – program cannot be run due to error in code

ii. Runtime/Logic error - occurs when a program is running/executing
   - i.e., something has gone wrong
   - e.g., User does something that is not anticipated by the program

Python provides the **try** and **except** statements to capture and handle error during the execution of a program.

- the **try** statement let you test a block of code for errors
- the **except** statement let you handle the error if an error, known as exception occurs

Exceptions

Finger Exercise:

```
try:
    print(x)
except:
    print("something goes wrong!")
```

But Python also provides more specific built-in exceptions that are raised when the corresponding errors occur.

Some common built-in exceptions:

- NameError Raised when a variable is not found
- ValueError Raised when a value is not valid
- KeyError Raised when a key is not found in a dictionary.

Python Error Handling

What could be the potential problem?

```
n = int(input("Enter a value"))
print(f"Value is {n}")
```

Using try-except

```
try:
    n = int(input("Enter a value: "))
    print(f"Value is {n}")
except ValueError:
    print(f"Please enter the value properly")
except:
    print("something goes wrong!")
```

Python Error Handling - `Else`

The `else` keyword can be used to define the code to be executed if no error is raised.

```
try:
    n = int(input("Enter a value: "))
except ValueError:
    print(f“Please enter the value properly“
except:
    print(“something goes wrong!")
else:
    print(f"Value is {n}")
```

Python Error Handling - `Finally`

The **`finally`** keyword can be used to define the code to be executed regardless of whether the **`try`** raises an error or not

```
try:
    n = int(input("Enter a value: "))
except ValueError:
    print(f"Please enter the value properly")
except:
    print("something goes wrong!")
else:
    print(f"Value is {n}")
finally:
    print("try-except completed")
```

Python File Error Handling

Another example

```
try:
    f = open(“file.txt)
    try:
      f.write(“Hello”)
    except:
      print(“Something goes wrong when writing to file")
    finally:
      f.close()

except:
    print("something goes wrong when opening the file")
```

Assert Statement

The Python assert statement is useful for debugging and checking your code during development

- it test whether an expression or a condition is true
  - and raise AssertionError and stop the program if false

Examples: You can use it to check values

```
assert 5+10 == 15   # this will pass silently
assert 5+10 == 12, "This is not correct"
assert len(numbers) > 1, "need more than 1 number"
```

And check the correctness of a function code

```
def divide(x,y):
    assert y!=0
    return x/y;

assert divide(10,5) == 2
```

Types of Exceptions in Python

Exception	Cause/Description
`AssertionError`	Failure of an `assert` statement
`AttributeError`	Failure of an attribute assignment or reference
`EOFError`	Raised when the `input()` function hits an end-of-file condition
`FloatingPointError`	Failure of a floating-point operation
`GeneratorExit`	Raised when a generator's `close()` method is called
`ImportError`	Raised when an import module is not found
`IndexError`	Raised when the index of a sequence is out of range
`KeyError`	Raised when a key is not found in a dictionary
`KeyboardInterrupt`	Raised when the user hits the interrupt key (*Ctrl* + *C* or *Delete*).
`MemoryError`	Raised when an operation runs out of memory
`NameError`	Raised when a variable is not found in local or global scope
`NotImplementedError`	Raised by abstract methods
`OSError`	Raised when system operation causes system-related error
`OverflowError`	Raised when the result of an arithmetic operation is too large to be represented
`ReferenceError`	Raised when a weak reference proxy is used to access a garbage-collected referent
`RuntimeError`	Raised when an error does not fall under any other category
`StopIteration`	Raised by a `next()` function to indicate that there is no further item to be returned by the iterator

Aside:
Software Profiling
(Will not be tested)

Profiling

Once our application program is tested and debugged

- we may want to improve/optimize its performance
  - e.g. make it run faster if it takes too long to execute,

Code profiling is a method to determine which line of code is consuming the most resources

- such as CPU processing time and memory
- to help identifying performance bottleneck

The are software tools that can be used to profile the programme code while the code is executing

- i.e., without having to modify the code

Software Profiling

One basic profiling technique is to use the '**`time`**' module

- which enable measurement of code execution time by using its timer functions:

  `time.perf_counter()` - the current time (wall-clock time)

  `time.process_time()` - the CPU time of the current process

```
def main():
    for function in sleep, add: # call the two functions in turn
        t1 = time.perf_counter(), time.process_time()
        function() # call sleep, then add
        t2 = time.perf_counter(), time.process_time()
        print(f"{function.__name__}()")
        print(f" Real time: {t2[0] - t1[0]:.2f} seconds")
        print(f" CPU time: {t2[1] - t1[1]:.2f} seconds")
```

Software Profiling (cont.)

```
def sleep():
    # this function goes into sleep without consuming any CPU time.
    # it is suspended for 1 sec, allow CPU to run another thread.
    time.sleep(1)

def add():
    for a in range(100_000_000):
        a += 1

main()
```

```
sleep()
 Real time: 1.00 seconds
 CPU time: 0.00 seconds
add()
 Real time: 4.62 seconds
 CPU time: 3.27 seconds
```