Chapter 9: Exceptions

Last Update: May 2, 2023

Program 1:

09 05-global.py

```
 1 def outer():
 2     def inner():
 3         print(f"            From inner: {x}")
 4     inner()
 5     print(f"        From outer: {x}")
 6
 7 x = 'global value'
 8 outer()
 9 print(f"    From global: {x}")
10 print()
11
12 # You can always use "global" keyword to force the variable
13 #     to have global scope
14 def outer2():
15     def inner2():
16         #global x
17         print(f"            From inner x = {x}")
18     x = 'enclosed value'
19     inner2()
20     print(f"        From outer x = {x}")
21
22 x = 'global value'
23 outer2()
24 print(f"    From global x = {x}")
25 print()
26
27 # You can always avoid the problem by never use the same
28 #    variable name
29 def outer3():
30     def inner3():
31         print(f"            From inner x = {x}")
32         print(f"            From inner y = {y}")
33     y = 'enclosed value'
34     inner3()
35     print(f"        From outer x = {x}")
36     print(f"        From outer y = {y}")
37
38 x = 'global value'
39 outer3()
40 print(f"    From global: {x}")
```

Program 2:

09	06-nonlocal.py

```
 1 # Nonlocal
 2 # It allows you to change a variable in an enclosed outer function
 3 #    to be changed inside an inner function
 4 # The same effect applies to Global
 5
 6 def next():
 7     input("> ")
 8
 9 def outer(): # outer function
10     print(f'    Entering outer')
11     x = 'value defined in outer'
12     y = 'value defined in outer'
13     print(f'    From outer x = {x}')
14     print(f'    From outer y = {y}')
15     next()
16
17     def inner(): # inner function
18         print(f'        Entering inner')
19         #nonlocal x                  # Turn on or off #1
20         global x                    # Turn on or off #2
21         x = 'value defined in inner'
22         y = 'value defined in inner'
23         print(f'        From inner x = {x}')
24         print(f'        From inner y = {y}')
25         next()
26
27     inner()
28     print(f'    From outer x = {x}')
29     print(f'    From outer y = {y}')
30
31 print(f'Entering global')
32 x = 'value defined in global'
33 y = 'value defined in global'
34 next()
35 outer()
36 next()
37 print(f'From global x = {x} ')
38 print(f'From global y = {y} ')
```

Program 3:

09 17-use before define.py

```
def next():
    input("Next> ")

def f():
    global s
    print(f'Inside  f(): s = {s}')
    # local variable 's' referenced before assignment
    s = 'Go Rockets'  # This made it Local

print("\tCase 1: s is global")
s = "Go Coogs"
f()
print(f'Outside f(): s = {s}') # global, unchanged
next()

print("\tCase 2: Does not work without global")
def f():
    print(f'Inside  f(): s = {s}')
    # local variable 's' referenced before assignment
    s = 'Go Rockets'  # This made it Local

s = "Go Coogs"
f()
print(f'Outside f(): s = {s}') # global, unchanged
```

Program 4:

09 20-nested function.py

```
def func1(): # outer function

    def func2(): # inner function
        print ("Hello from inner")

    print("Hello from outer")
    func2()

func1()
```

Program 5:

09 22-nested exam.py

```
 1 # This is an example of nested functions
 2 # function get_int() is defined inside func()
 3 # and can only be accessed inside func().
 4 # Try un-comment the get_int(9) and see what happens.
 5 def func(num):
 6     num_list = []
 7     print(f'Get {num} integer numbers.')
 8
 9     def get_int(n):
10         return abs(int(input(f'Enter integer #{n}: ')))
11
12     for i in range(num):
13         num_list.append(get_int(i+1))
14     return(num_list)
15
16 print(func(5))
17 #get_int(9)
```

Program 6:

09 26-nonlocal.py

```
 1 # If you assign a value to a variable x anywhere in a function,
 2 #                                        ^^^^^^^^
 3 #    x is a local variable
 4 #    UNLESS you declare x to be NONLOCAL or GLOBAL
 5
 6 def outer():
 7     def inner():
 8         nonlocal x                    # Turn on and off #2
 9         #print("        inner:", x)   # Turn on and off #1
10         x = 'defined in inner'
11         print("        inner:", x)
12
13     x = 'defined in outer'
14     print("    outer:", x)
15     inner()
16     print("    outer:", x)
17
18 # main
19 x = 'defined in main'
20 print('main: ', x)
21 outer()
22 print('main: ', x)
```

Program 7:

09 30-immutable para.py

```
def next():
    input("\n> ")

# This shows the same list being passed from main program
# to the function.
def func1(s):
    print(id(s), s)

# A parameter (s in this case) is local
# Changing local variables does not change the actual parameter.
# S is a pointer, it just points to another memory location.
# See the new id.
def func2(s):
    print(id(s), s)
    s = [66, 77, 88, 99]
    print(id(s), s)

# The s points to a list which is mutable.
# So, we can change the list directly. Note the address remain the
# same before and after the call.
def func3(s):
    print(id(s), s)
    s[0] = 111
    s.append(999)
    print(id(s), s)

mylist = [1, 2, 3]
print("\tCase 1: List Parameter")
print(id(mylist), mylist)
func1(mylist)
print(id(mylist), mylist)
next()

print("\tCase 2: Parameter Changed")
func2(mylist)
print(id(mylist), mylist)
next()

print("\tCase 3: Update list")
func3(mylist)
print(id(mylist), mylist)
```

Program 8:

09 31-mutable para.py

```
def change1(alist):
    print('   ', id(alist), alist)
    alist[0] = 'Watermelon'
    alist.append('Pear')
    print('   ', id(alist), alist)

def change2(alist):
    print('   ', id(alist), alist)
    alist = ['Mango', 'banana', 'cherry', 'Pear']
    print('   ', id(alist), alist)

mylist = ['apple', 'banana', 'cherry']
print(id(mylist), mylist)
change1(mylist)
print(id(mylist), mylist)
change2(mylist)
print(id(mylist), mylist)
```

Program 9:

09 40-passing a list.py

```
def foo(list):
    print('   inside', list)
    list.append(99)
    list[2] = -1
    print('   inside', list)

s = [1, 2, 3, 4, 5]
print('Before   ', s)
foo(s)
print('After    ', s)
```

Program 10:

09 41-call by value.py

```
def test_para(a_string):
    print("    Received  :", a_string)
    a_string = "That is the question"
    print("    Changed to:", a_string)

a_string = "To be or not to be"

print ("Before test:   ", a_string)
test_para(a_string)
print ("After  test:   ", a_string)

# What if you really want to change the string?
```

Program 11:

09 42-wrapper.py

```
def test_para(a_string):
    print ("    Received  :", a_string)
    a_string[0] = "That is the question"
    print ("    Changed to:", a_string)

a_string = "To be or not to be"
wrapper = [a_string]

print (f"Before test:   {a_string} {id(a_string)}, {wrapper},
{id(wrapper)}")
test_para(wrapper)
a_string = wrapper[0]
print (f"After  test:   {a_string} {id(a_string)}, {wrapper},
{id(wrapper)}")
```

Program 12:

09 50-function as argument.py

```
def foo(f, para):
    print(f"Calling {f}(\"{para}\") inside foo().")
    f(para)

def bar(para):
    print(f"Inside bar(\"{para}\").")

bar("Hello world!")
foo(bar, "Howdy")
```

Program 13:

09 52-key function.py

```
def norm(s):
    return s.casefold()

def last(s):
    return s[-1]

def len_norm(s):
    return(len(s), s.casefold())

fruits = ['cherry', 'banana', 'Apple', 'Pear', 'Watermelon', 'peach']
print(sorted(fruits))
print(sorted(fruits, key=norm))
print(sorted(fruits, key=last))
print(sorted(fruits, key=len))
print(sorted(fruits, key=len_norm))
print(max(fruits))
print(max(fruits, key=norm))
```

Program 14:

```
09 60-default arg.py
 1 def cost(unitCost, length, width=6.5, depth=0.5):
 2     return unitCost*length*width*depth
 3
 4 print(cost(100, 10, 8, 1))
 5 print(cost(100, 10, 2))
 6 print(cost(100, 10))
 7 #print(cost(100))
```

Program 15:

```
09 70-fibonacci.py
 1 def fib(n):
 2     print("fib(",n,")", sep='')
 3     if n==0:
 4         return 0
 5     elif n==1:
 6         return 1
 7     else:
 8         return fib(n-1)+fib(n-2)
 9
10 print(fib(25))
```

Program 16:

```
09 80-lambda.py
 1 # The first example shows sorting strings
 2 # Case-sensitive!
 3 colors = ["Goldenrod", "purple", "Salmon", "turquoise", "cyan"]
 4 print(colors)
 5
 6 print(sorted(colors, key=lambda elem: elem.casefold()))
 7 print('Case-sensitive Test'.casefold())
 8
 9 # The second one shows various ways to sort the same list
10 list1 = [(2,'B'), (5,'X'), (4,'D'), (2,'A'), (3,'C'), (4,'X'),
11 (4,'A')]
12 print("Original")
13 print(list1)
14
15 print("\nsort, default")
16 print(sorted(list1))
17
18 print("\nsort, by letter")
19 print(sorted(list1, key=lambda row: row[1]))
20
21 print("\nsort, by number+letter")
22 print(sorted(list1, key=lambda row: (row[0],row[1])))
```

Program 17:

09 82-lambda 2.py

```
mylist = [
    [10, 20, 30],
    [13, 20, 16],
    [15, 39, 43],
    [11, 12, 13]
]
# Method 1: define a function to sort on the value
# for each element
def tiny(row):
    return row[0]+row[2]

print("Without Lambda")
mylist.sort(key=tiny)
print(mylist)
#
# Method 2: define the function using lambda
#
print("\nWith Lambda")
mylist.sort(key=lambda row:row[0]+row[2])
print(mylist)
#
# Same result
```