ISE 101 – Introduction to Information Systems

- Lecture 6 Objectives:
  - Exceptions
  - Recursive functions
  - Tuples
  - Formatted printing

EXCEPTIONS

Exceptions

- Exceptions are typically rare errors that interrupt/stop the execution of the program
  - Divison by zero
  - Wrong type of value entered by the user
  - Trying to open a file that does not exist
  - …
- Handling exceptions is critically important for
  - commercial software products
  - mission-critical software (healthcare, defence etc.)

Handling Exceptions

- In order to handle an exception, the risky code portion should be written inside a try/except statement
- Structure

```
try:
    statement 1
    statement 2
    statement 3
    ...
except [exception type]:
    statement 1
    statement 2
    ...
else:
    statement 1
    statement 2
    ...
```

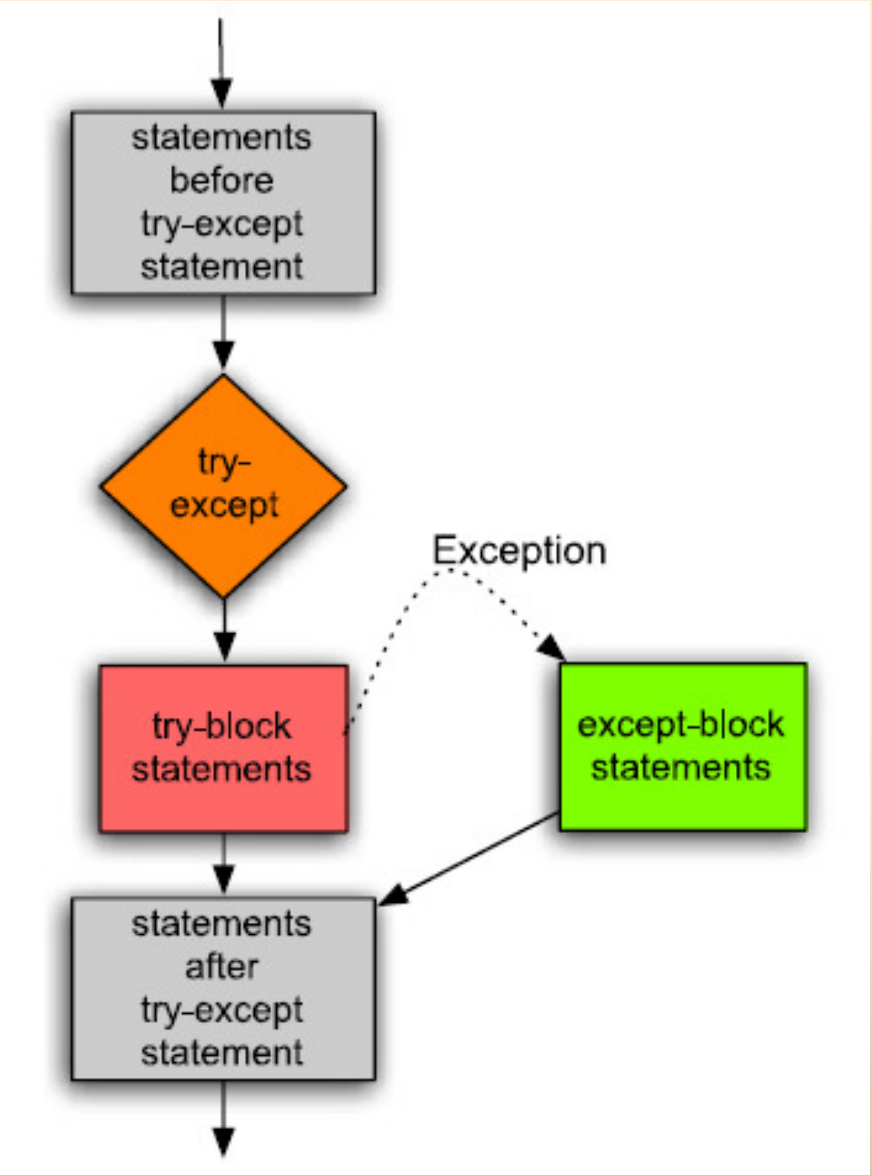
Risky code is placed inside the scope of try

exception type to be catched

If an exception occurs inside the scope of try, statements within the scope of except are executed.

If no exceptions occur, statements in the scope of else are executed

Handling Exceptions



Example

```
x=input('Enter an integer: ')
x=int(x)
```

```
Enter an integer: 3.4
Traceback (most recent call last):
  File "None", line 2, in <module>
builtins.ValueError: invalid literal for int() with base 10: '3.4\r'
```

Example

```
try:
   x=input('Enter an integer: ')
   x=int(x)
except ValueError:
   print('Entered value should be an integer')
else:
   print(str(x) + ' is entered')
```

```
Enter an integer: 3.4
Entered value should be an integer
```

```
Enter an integer: 5
5 is entered
```

Multiple Exceptions

- If there is possibility of many exceptions each can be handled separately with an appropriate message

```
try:
    x=input('Enter an integer: ')
    result=10/int(x)
except ValueError:
    print('Entered value should be an integer')
except ZeroDivisionError:
    print('Enter a nonzero value')
else:
    print(str(res))
```

Error Types

- There are too many error types that can be handled
- A comprehensive list is given at

http://docs.python.org/library/exceptions.html

Important Exception Errors

- **EOFError**: Raised when one of the built-in functions (**input()** or **raw_input()**) hits an end-of-file condition (EOF) without reading any data.

  When you ask for an input from the user and user does not input anything/press enter directly

- **IOError**: Raised when an I/O operation (such as a **print** statement, the built-in **open()**function or a method of a file object) fails for an I/O-related reason, e.g., ``file not found'' or ``disk full''.

Important Exception Errors

- **ImportError**: Raised when an **import** statement fails to find the module definition or when a **from...import** fails to find a name that is to be imported.
- **KeyboardInterrupt**: Raised when the user hits the interrupt key (normally Control-C or Delete).
- **MemoryError**: Raised when an operation runs out of memory.
- **OverflowError**: Raised when the result of an arithmetic operation is too large to be represented.
- **ZeroDivisionError**: Raised when the second argument of a division or modulo operation is zero.

Catching All Exceptions

- If no error type is given after except, all exceptions are catched.
- However, there is no way to know what specifically happened.

```
try:
    x=input('Enter an integer: ')
    result=10/int(x)
except:
    print(`Something went wrong!')
else:
    print(str(res))
```

Common Statements

- If there are statements that has to be executed both in the case exception or no-exceptions, they are placed under finally statement.

```
try:
        statement 1
        statement 2
        statement 3
        ...
except [exception type]:
        statement 1
        statement 2
        ...
finally:
        statement 1
        statement 2
        ...
```

These statements are executed before leaving try/except block

Example

```
try:
    f = open('poem.txt')
    for line in f:
        # do something
except:
    print('Something went wrong during file IO!')
finally:
    f.close()
    print('(Cleaning up: Closed the file)')
```

Raising Exceptions

- It is possible to raise exceptions from your code
- This is typically required if you are writing a library that will be reused

```
>>> raise NameError('Message')
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
builtins.NameError: Message
```

RECURSIVE FUNCTIONS

Recursive Functions

- Recursive functions are a specific type of functions that calls itself from its body.
- Recursive functions are short however they are hard to debug
- A recursive function has 3 parts:
  - Check for end of the recursion (If this step is skipped an infinite recursion will continue until stack overflow)
  - Processing data, computation etc. (body) and calling itself
  - Return result

Example

- Factorial computation
- n!=n*(n-1)*(n-2)…3.2.1
- n!=n*(n-1)!

```
def fact(n):
    if n<=1:
        return 1

    result=n*fact(n-1)

    return result
```

Example

- Write a recursive function that takes a string as its argument and reverses the string

```
def reverse_str(string):
    if len(string)<=1:
        return string

    result=reverse_str(string[1:])+string[0]

    return result
```

Example

- Write a recursive function that computes the *nth Fibonacci number.*
- The Fibonacci numbers are defined as follows: Fib*(0) = 1, Fib(1) = 1, Fib(n) =* Fib*(n – 1) + Fib(n – 2).*

```
def fib(number):
    if number<2:
        return 1

    result=fib(number-1)+fib(number-2)

    return result
```

TUPLES

Tuples

- Tuple is a Python type that holds multiple objects together
- Tuples are similar to lists
  - They have less features compared to lists
  - They are immutable: Once formed, they cannot be modified (like strings)
- Tuples are formed using paranthesis (lists are formed by square brackets)
- For example:

```
>>> t = (1.0, 3.4, 'ITU')
>>> type(t)
<class 'tuple'>
```

Tuples

- To create a tuple with a single element (called singleton), you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

- Without the comma, Python treats ('a') as a string in parentheses:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Tuples

- Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

  >>> t = tuple()

  >>> print t

  ()
- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

  >>> t = tuple('lupins')

  >>> print t

  ('l', 'u', 'p', 'i', 'n', 's')

Tuples

- Most list operators also work on tuples.
- The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

- And the slice operator selects a range of elements.

```
>>> print t[1:3]
('b', 'c')
```

Tuples

- But if you try to modify one of the elements of the tuple, you get an error:

  >>> t[0] = 'A'

  TypeError: object doesn't support item assignment
- You can't modify the elements of a tuple, but you can replace one tuple with another:

  >>> t = ('A',) + t[1:]

  >>> print t

  ('A', 'b', 'c', 'd', 'e')

Tuple assignment

- It is often useful to swap the values of two variables.
- With conventional assignments, you have to use a temporary variable.
- For example, to swap a and b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

- This solution is cumbersome; tuple assignment is more elegant:

```
>>> a, b = b, a
```

- The left side is a tuple of variables; the right side is a tuple of expressions.
- Each value is assigned to its respective variable.

```
>>> t1=tuple('ITU')
>>> t2=tuple('YTU')
>>> t1,t2=t2,t1
>>> print t1
('Y','T','U')
>>> print t2
('I','T','U')
```

Tuples and Lists

- If the sequences are not the same length, the result gets the length of the shorter one.

```
>>> a=list(zip('ITU','12345'))
>>> print(a)
[('I', '1'), ('T', '2'), ('U', '3')]
```

- You can use tuple assignment to traverse a list of tuples:

```
for letter,number in a:
        print(letter , number)
I 1
T 2
U 3
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number.

Example

- Write a Python function that takes 2 arguments, compares the elements of two tuples and returns True if their corresponding elements have the same value
- For example,

  has_match(t1,t2) should return
  - True for t1=(1,2,4) and t2=(4,2,7)
  - False for t1(3,4,7) and t2=(2,23,5)

Example

```
def has_match(t1,t2):
    for e1,e2 in zip(t1,t2):
        if e1==e2:
            return True

    return False
```

Sorting Tuples

- The comparison operators work with tuples and other sequences;
- Python starts by comparing the first element from each sequence.
- If they are equal, it goes on to the next elements, and so on, until it finds elements that differ.
- Subsequent elements are not considered (even if they are really big).

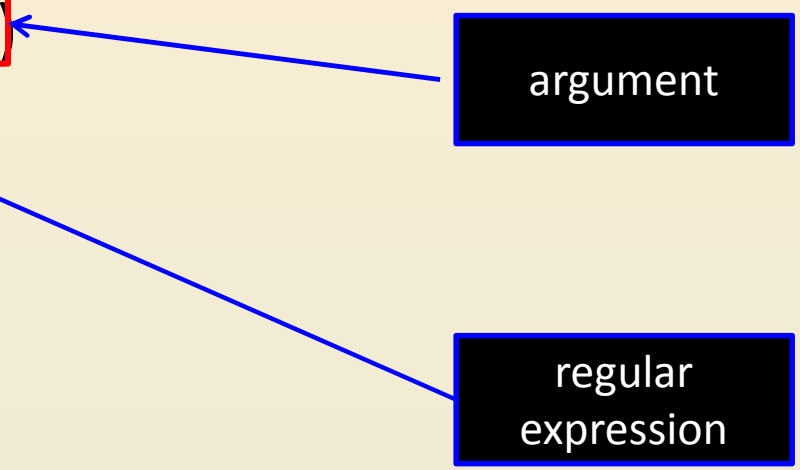
```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

FORMATTED PRINTING

Formatted Output

print (‘<regular expression>’ % (argument1, argument2))

```
>>> print ('result: %05d' %5)
result: 00005
>>> print('result: %d'%5)
result: 5
```



Formatted Output

- Regular expression

%<alignment flag><minimum length>.<precision length><type>

<type> → mandatory field. Shows the type of the output

Formatted Output

'd' → Signed integer decimal
'i' → Signed integer decimal.

```
>>> print ('%d %d' % (4,5))
4 5
>>> print('%d' % 4.56)
4
>>> print ('%i' % 4.56)
4
>>> print ('%i' % -4.56)
-4
>>> print ('%i' % -4.1)
-4
```

Formatted Output

'o' → Signed octal value.

```
>>> print ('%o' % 4)
4
>>> print ('%o' % 14)
16
>>> print ('%o' % 9)
11
>>> print ('%o' % -8)
-10
```

Formatted Output

'x' → Signed hexadecimal (lowercase).

'X' → Signed hexadecimal (uppercase).

```
>>> print( '%x' % 15)
f
>>> print ('%x' % -15)
-f
>>> print ('%X' % -15)
-F
```

Formatted Output

'e' → Floating point exponential format (lowercase).

'E' → Floating point exponential format (uppercase).

```
>>> print( '%e' % -15.34)
-1.534000e+01
>>> print ('%E' % .873246286424 34)
8.732463E-01
```

Formatted Output

'f' → Floating point decimal format.

'F' →Floating point decimal format.

```
>>> print ('%f' % -15.34)
-15.340000
>>> print ('%F' % -15.34)
-15.340000
```

Formatted Output

's' → String

```
>>> print ('Isim: %s'%'Hasan‘)
Isim: Hasan
```

Formatted Output

%<alignment flag><minimum length>.<precision length><type>

Alignment flag:

'0' →The conversion will be zero padded for numeric values.

'-' → The converted value is left adjusted (overrides the '0' conversion if both are given).

' ' → (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

'+' → A sign character ('+' or '-') will precede the conversion (overrides a “space” flag).

Formatted Output

%<alignment flag><minimum length>.<precision length><type>

Alignment flag:

```
>>> print ('%05d'%4)
00004
>>> print('% 5d'%4)
    4
>>> print ('%-5d'%4)
4
>>> print ('%+5d'%4)
   +4
>>> print ('%+5d'%-4)
   -4
```

Example

- Write print expressions such that the output would look like:

```
    5
   4
  3
 2
1
```

```
for i in range(5,0,-1):
    string='%'+str(i)+'d\n'
    print(string%i)
```

Example

```
>>> import math
>>> print(math.pi)
3.14159265359
```

Write print expressions such that the output would look like:

```
Pi number: 3.1
Pi number: 3.14
Pi number: 3.141
Pi number: 3.1415
Pi number: 3.14159
```

```
import math

for i in range(1,6):
    exp='%.'+str(i)+'f\n'
    print(exp%math.pi)
```