ITP 115 – Programming in Python

Errors

Review

Input

- The **input** function in Python *always* returns a string even when we want the user to enter a number.
- We taught you to use the **int** function to convert the string to an integer.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
```

Bad Input

- What if the user doesn't enter a number?

```
age = int(input("Enter your age: "))
```


```
Enter your age: twenty
Traceback (most recent call last):
  File "../Errors.py", line 6, in <module>
    age = int(input("Enter your age: "))
ValueError: invalid literal for int() with base 10: 'twenty'
```

String Error Checking Methods

- `string` is a variable holding a string

Method	Description
`string.isalnum()`	Returns **`True`** if **`string`** contains only letters and numbers Returns **`False`** otherwise
`string.isalpha()`	Returns **`True`** if **`string`** contains only letters Returns **`False`** otherwise
`string.isdigit()`	Returns **`True`** if **`string`** contains only digits Returns **`False`** otherwise
`string.isspace()`	Returns **`True`** if **`string`** contains only whitespace Returns **`False`** otherwise

Example – isdigit

- Use the **isdigit** method to make sure the user enters a number.

```
ageStr = input("Enter your age: ")
while not ageStr.isdigit():
   ageStr = input("Enter a number for your age: ")

age = int(ageStr)
```

Error Handling

- Those approaches use `if` to check a condition **BEFORE** an operation is performed

- Another approach is to perform an operation **FIRST** and then handle any errors **AFTER**

What will happen?

```
numStr = input("Enter a numerator: ")
while not numStr.isdigit():
    numStr = input("Enter a number for the numerator: ")
num = int(numStr)

denStr = input("Enter a denominator: ")
while not denStr.isdigit():
    denStr = input("Enter a number for the denominator: ")
den = int(denStr)

result = num/den
```

```
Enter a numerator: 42
Enter a denominator: 0
```

Error

```
result = num/den
```

```
Enter a numerator: 42
Enter a denominator: 0
Traceback (most recent call last):
  File "../Errors.py", line 18, in <module>
    result = num / den
ZeroDivisionError: division by zero
```

- Even if we use the string methods, we may still get an error.

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called **exceptions** and are not unconditionally fatal.

Handling Exceptions

- When Python runs into an error, it stops the current program and displays an error message
  - It raises an exception.

- If nothing is done with the exception, Python halts what it's doing and prints an error message.

- Most basic way to handle (or trap) exceptions is to use the **`try`** statement with an **`except`** clause.

Understanding Exceptions

```
3/0
```

This code causes an error (division by zero)

```
Traceback (most recent call last):
  File "../Errors.py", line 1, in <module>
    3/0
ZeroDivisionError: division by zero
```

Program crashes (stops)

Understanding Exceptions

```
try:
   3/0
```

IF

This code causes an error

```
except:
   print("Oops! Divided by zero")
```

THEN

Run this code

Program continues running with no error message.

"Exits gracefully"

Common Exception Types

Exception Type	Description
`IOError`	Raised when an I/O operation fails, such as when an attempt is make to open a nonexistent file in read mode.
`IndexError`	Raised when a sequence is indexed with a number of a nonexistent element.
`KeyError`	Raised when a dictionary key is not found.
`NameError`	Raised when a name (of a variable or function, for example) is not found.
`SyntaxError`	Raised when a syntax error is encountered.
`TypeError`	Raised when a built-in operation or function is applied to an object of inappropriate type.
`ValueError`	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero.

Specifying Exceptions

```
try:
   #code

except ExceptionType:
   #error handling code
```

This checks **ONLY** for the specific ***ExceptionType*** listed

Specifying Exceptions

```
try:
  3/0

except ZeroDivisionError:
  print("Oops! You divided by zero")
```

This checks **ONLY** for **ZeroDivisionError**

Specifying Multiple Exceptions (method 1)

```
try:
   #code


except (ExceptionType1, ExceptionType2):
   #error handling code
```

This code runs if **EITHER** ***ExceptionType1*** **OR** ***ExceptionType1*** occurs

Specifying Multiple Exceptions (method 2)

```
try:
   #code

except ExceptionType1:
   #error handling code
```

This code runs **ONLY IF** ***ExceptionType1*** occurs

```
except ExceptionType2:
   #error handling code
```

This code runs **ONLY IF** ***ExceptionType2*** occurs

Exception Arguments

- You can display the standard error message when an exception occurs

```
try:
   3/0

except ValueError as e:
   print("The error msg was " + str(e))
```

Else Clause

- After **`try / except`**, you can use **`else`**
- The **`else`** block runs ONLY IF no exceptions occurred

```
try:
    x = 3/2

except ZeroDivisionError:
    print("Oops! Division by zero")

else:
    print("Division success!")
```

Execution Flow with Exceptions

1. Run statements in **`try`**

2. If no errors
   a. Skip **`except`** and run **`else`**

3. If a statement in **`try`** raises exception
   a. Skip the remaining statements in **`try`**
   b. Check for any **`except`** block that matches the raised exception
   c. If no matching **`except`** block, exit program

Execution Flow with Exceptions

Start

ZeroDivisionError

Error matched

Continue

```
print("start program")

try:
    num = 3/0
    print("Math is fun!")

except ZeroDivisionError:
    print("That was not a number!")

else:
    print("Division success!")

print("end program")
```