10.1 Handling exceptions using try and except

Error-checking code is code that a programmer introduces to detect and handle errors that may occur while the program executes. Python has special constructs known as ***exception-handling*** constructs because they handle *exceptional circumstances*, another word for errors during execution.

Consider the following program that has a user enter weight and height, and that outputs the corresponding body-mass index (BMI is one measure used to determine normal weight for a given height).

Figure 10.1.1: BMI example without exception handling.

```
user_input = ''
while user_input != 'q':
    weight = int(input("Enter
weight (in pounds): "))
    height = int(input("Enter
height (in inches): "))

    bmi = (float(weight) /
float(height * height)) * 703
    print('BMI:', bmi)
    print('(CDC: 18.6-24.9
normal)\n')
    # Source www.cdc.gov

    user_input = input("Enter any
key ('q' to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): One-hundred
fifty
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    weight = int(input("Enter weight (in
pounds): "))
ValueError: invalid literal for int()
with base 10: 'One-hundred fifty'
```


Above, the user entered a weight by writing out "One-hundred fifty", instead of giving a number such as "150", which caused the int() function to produce an exception of type ValueError. The exception causes the program to terminate.

Commonly, a program should gracefully handle an exception and continue executing, instead of printing an error message and stopping completely. Code that potentially may produce an exception is placed in a ***try*** block. If the code in the try block causes

an exception, then the code placed in a following ***except*** block is executed. Consider the program below, which modifies the BMI program to handle bad user input.

Figure 10.1.2: BMI example with exception handling using try/except.


```
user_input = ''
while user_input != 'q':
    try:
        weight = int(input("Enter weight
(in pounds): "))
        height = int(input("Enter height
(in inches): "))

        bmi = (float(weight) /
float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')
# Source www.cdc.gov
    except:
        print('Could not calculate health
info.\n')

    user_input = input("Enter any key ('q'
to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): One-
hundred fifty
Could not calculate health
info.

Enter any key ('q' to quit): a
Enter weight (in pounds): 200
Enter height (in inches): 62
BMI: 36.57648283038502
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): q
```

The try and except constructs are used together to implement ***exception handling***, meaning handling exceptional conditions (errors during execution). A programmer could add additional code to do their own exception handling, e.g., checking if every character in the user input string is a digit, but such code would make the original program difficult to read.

Construct 10.1.1: Basic exception handling constructs.


```
try:
    # ... Normal code that might produce errors
except: # Go here if <i>any</i> error occurs in try block
    # ... Exception handling code
```

PARTICIPATION

ACTIVITY 10.1.1: How try and except blocks handle exceptions.

Animation captions:

1. When a try is reached, the statements in the try block are executed.
2. Any statements in the try block not executed before the exception occurred are skipped.

When a try is reached, the statements in the try block are executed. If no exception occurs, the except block is skipped and the program continues. If an exception does occur, the except block is executed, and the program continues *after* the try block. Any statements in the try block not executed before the exception occurred are skipped.

PARTICIPATION ACTIVITY 10.1.2: Exception basics.

1) Execution jumps to an except block only if an error occurs in the preceding try block.
   - ○ True
   - ○ False

2) After an error occurs in a try block, and the following except block has executed, execution resumes after where the error occurred in the try block.
   - ○ True
   - ○ False

10.2 Multiple exception handlers

Sometimes the code in a try block may generate different types of exceptions. In the previous BMI example, a ValueError was generated when the int() function was passed

a string argument that contained letters. Other types of errors (such as NameError, TypeError, etc.) might also be generated, and thus a program may need to have unique exception handling code for each error type. Multiple ***exception handlers*** can be added to a try block by adding additional except blocks and specifying the specific type of exception that each except block handles.

Construct 10.2.1: Multiple except blocks.

```
try:
    # ... Normal code
except exceptiontype1:
    # ... Code to handle exceptiontype1
except exceptiontype2:
    # ... Code to handle exceptiontype2
...
except:
    # ... Code to handle other exception types
```

PARTICIPATION ACTIVITY 10.2.1: Multiple exception handlers.

Animation captions:

1. Multiple exception handlers can be added to a try block by adding additional except blocks and specifying the particular type of exception that each except block handles.

An except block with no type (as in the above BMI example) handles any unspecified exception type, acting as a catch-all for all other exception types. Good practice is to generally *avoid* the use of a catch-all except clause. A programmer should instead specify the particular exceptions to be handled. Otherwise, a program bug might be hidden when the catch-all except clause handles an unexpected type of error.

If no exception handler exists for an error type, then an ***unhandled exception*** may occur. An unhandled exception causes the interpreter to print the exception that occurred and then halt.

The following program introduces a second exception handler to the BMI program, handling a case where the user enters "0" as the height, which would cause a ZeroDivisionError exception to occur when calculating the BMI.

Figure 10.2.1: BMI example with multiple exception types.

```
user_input = ''
while user_input != 'q':
    try:
        weight = int(input("Enter weight
(in pounds): "))
        height = int(input("Enter height
(in inches): "))

        bmi = (float(weight) /
float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')
# Source www.cdc.gov
    except ValueError:
        print('Could not calculate health
info.\n')
    except ZeroDivisionError:
        print('Invalid height entered.
Must be > 0.')

    user_input = input("Enter any key ('q'
to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): One-
hundred fifty
Could not calculate health
info.

Enter any key ('q' to quit): a
Enter weight (in pounds): 150
Enter height (in inches): 0
Invalid height entered. Must be
> 0.
Enter any key ('q' to quit): q
```


In some cases, multiple exception types should be handled by the same exception handler. A tuple can be used to specify all of the exception types for which a handler's code should be executed.

Figure 10.2.2: Multiple exception types in a single exception handler.

```
try:
    # ...
except (ValueError, TypeError):
    # Exception handler for any ValueError or TypeError that occurs.
except (NameError, AttributeError):
    # A different handler for NameError and AttributeError exceptions.
except:
    # A different handler for any other exception type.
```


PARTICIPATION ACTIVITY 10.2.2: Multiple exceptions.

1) Fill in the missing code so that any type of error in the try block is handled.

```
ages = []
prompt = "Enter age ('q' to
quit):"
user_input = input(prompt)
while user_input != 'q':
      try:

ages.append(int(user_input))
            user_input =
input(prompt)
      
:
            print('Unable to
add age.')
print(ages)
```

Check **Show answer**

2) An AttributeError occurs if a function does not exist in an imported module. Fill in the missing code to handle AttributeErrors gracefully and generate an error if other types of exceptions occur.

```
import my_lib
try:
      result =
my_lib.magic()
                        :
      print('No magic()
function in my_lib.')
```

Check **Show answer**

3) If a file cannot be opened, then an IOError may occur. Fill in the missing code so that the

program specially handles AttributeErrors and IOErrors, and also doesn't crash for any other type of error.

```
import my_lib
try:
      result =
my_lib.magic()
      f = open(result,
'r')
      print f.read()
                        :
      print('Could not
open file.')
except AttributeError:
      print('No magic()
function in my_lib')
except:
      print('Something
bad has happened.')
```

Check Show answer

Exploring further:

- Python built-in exception types

10.3 Raising exceptions

Consider the BMI example once again, in which a user enters a weight and height, and that outputs the corresponding body-mass index. The programmer may wish to ensure that a user enters only valid heights and weights, i.e., greater than 0. Thus, the programmer must introduce error-checking code.

A naive approach to adding error-checking code is to intersperse if-else statements throughout the normal code. Of particular concern is the yellow-highlighted code,

which is new branching logic added to the normal code, making the normal code flow of "get weight, get height, then print BMI" harder to see. Furthermore, the second check for negative values before printing the BMI is redundant and ripe for a programming error caused by inconsistency with the earlier checks (e.g., checking for <= here rather than just <).

Figure 10.3.1: BMI example with error-checking code but without using exception-handling constructs.

```
user_input = ''
while user_input != 'q':
    weight = int(input('Enter weight (in pounds): '))
    if weight < 0:
        print('Invalid weight.')
    else:
        height = int(input('Enter height (in inches): '))
        if height <= 0:
            print('Invalid height')

    if (weight < 0) or (height <= 0):
        print('Cannot compute info.')
    else:
        bmi = (float(weight) / float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')  # Source www.cdc.gov

    user_input = input("Enter any key ('q' to quit): ")
```

The following program shows the same error-checking carried out using exception-handling constructs. The normal code is enclosed in a try block. Code that detects an error can execute a ***raise*** statement, which causes immediate exit from the try block and the execution of an exception handler. The exception handler prints the argument passed by the raise statement that brought execution there. The key thing to notice is that the normal code flow is not obscured via new if-else statements. You can clearly see that the flow is "get weight, get height, then print BMI".

Figure 10.3.2: BMI example with error-checking code that raises exceptions.

```
user_input = ''
while user_input != 'q':
    try:
        weight = int(input('Enter weight (in
pounds): '))
        if weight < 0:
            raise ValueError('Invalid weight.')

        height = int(input('Enter height (in
inches): '))
        if height <= 0:
            raise ValueError('Invalid height.')

        bmi = (float(weight) / float(height *
height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')
        # Source www.cdc.gov

    except ValueError as excpt:
        print(excpt)
        print('Could not calculate health
info.\n')

    user_input = input("Enter any key ('q' to
quit): ")
```

```
Enter weight (in pounds):
166
Enter height (in inches): 55
BMI: 0.37885885885885884
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit):
a
Enter weight (in pounds):
180
Enter height (in inches): -5
Invalid height.
Could not calculate health
info.

Enter any key ('q' to quit):
a
Enter weight (in pounds): -2
Invalid weight.
Could not calculate health
info.

Enter any key ('q' to quit):
q
```


A statement like `raise ValueError('Invalid weight.')` creates a new exception of type ValueError with a string argument that details the issue. The programmer could have specified any type of exception in place of ValueError, e.g., NameError or TypeError, but ValueError most closely describes the exception being handled in this case. The ***as*** keyword binds a name to the exception being handled. The statement `except ValueError as excpt` creates a new variable excpt that the exception handling code might inspect for details about the exception instance. Printing the exception prints the string argument passed to the exception when raised.

PARTICIPATION ACTIVITY 10.3.1: Exceptions.

except: **try** **raise ValueError** **except NameError:**

except (ValueError, NameError):

Describes a block of code that uses exception-handling

An exception handler for NameError exceptions

An exception handler for ValueError *and* NameError exceptions

A catch-all exception handler

Causes a ValueError exception to occur

Reset

10.4 Exceptions with functions

The power of exceptions becomes even more clear when used within functions. If an exception is raised within a function and is not handled within that function, then the function is immediately exited and the calling function is checked for a handler, and so on up the function call hierarchy. The following program illustrates. Note the clarity of the normal code, which obviously "gets the weight, gets the height, and prints the BMI" – the error checking code does not obscure the normal code.

Figure 10.4.1: BMI example using exception-handling constructs along with functions.


```
def get_weight():
    weight = int(input('Enter weight (in pounds):
'))
    if weight < 0:
        raise ValueError('Invalid weight.')
    return weight

def get_height():
    height = int(input('Enter height (in inches):
'))
    if height <= 0:
        raise ValueError('Invalid height.')
    return height

user_input = ''
while user_input != 'q':
    try:
        weight = get_weight()
        height = get_height()

        bmi = (float(weight) / float(height *
height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')
        # Source www.cdc.gov

    except ValueError as excpt:
        print(excpt)
        print('Could not calculate health
info.\n')

    user_input = input('Enter any key ('q' to
quit): ')
```

```
Enter weight (in pounds):
150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit):
a
Enter weight (in pounds): -1
Invalid weight.
Could not calculate health
info.

Enter any key ('q' to quit):
a
Enter weight (in pounds):
150
Enter height (in inches): -1
Invalid height.
Could not calculate health
info.

Enter any key ('q' to quit):
q
```

Suppose get_weight() raises an exception of type ValueError. The get_weight() function does not handle exceptions (there is no try block in the function) so it immediately exits. Going up the function call hierarchy returns execution to the global scope script code, where the call to get_weight() was in a try block, so the exception handler for ValueError is executed.

Notice the clarity of the script's code. Without exceptions, the get_weight() function would have had to somehow indicate failure, perhaps through a special return value like -1. The script would have had to check for such failure and would have required additional if-else statements, obscuring the functionality of the code.

PARTICIPATION ACTIVITY 10.4.1: Exceptions in functions.

1) For a function that may contain a raise statement, the function's

statements must be placed in a try block within the function.

- ○ True
- ○ False

2) A raise statement executed in a function automatically causes a jump to the last return statement found in the function.

- ○ True
- ○ False

3) A key goal of exception handling is to avoid polluting normal code with distracting error-handling code.

- ○ True
- ○ False

10.5 Using finally to cleanup

Commonly a programmer wants to execute code regardless of whether or not an exception has been raised in a try block. For example, consider if an exception occurs while reading data from a file – the file should still be closed using the file.close() method, no matter if an exception interrupted the read operation. The ***finally*** clause of a try statement allows a programmer to specify *clean-up* actions that are always executed. The following illustration demonstrates.

PARTICIPATION ACTIVITY 10.5.1: Clean-up actions in a finally clause are always executed.

Animation captions:

1. If no exception occurs, then execution continues in the finally clause and then proceeds with the rest of the program.
2. If a handled exception occurs, then an exception handler executes and then the finally clause executes.

The finally clause is always the last code executed before the try block finishes.

- If *no exception* occurs, then execution continues in the finally clause, and then proceeds with the rest of the program.
- If a *handled exception* occurs, then an exception handler executes and then the finally clause.
- If an *unhandled exception* occurs, then the finally clause executes and then the exception is re-raised.
- The finally clause also executes if any break, continue, or return statement causes the try block to be exited.

The finally clause can be combined with exception handlers, provided that the finally clause comes last. The following program attempts to read integers from a file. The finally clause is always executed, even if some exception occurs when reading the data (such as if the file contains letters, thus causing int() to raise an exception, or if the file does not exist).

Figure 10.5.1: Clean-up actions using finally.

```
Enter file name: myfile.txt
Opening myfile.txt
Closing myfile.txt
Numbers found: 5 423 234
...
Enter file name: myfile.txt
Opening myfile.txt
Could not read number from
myfile.txt
Closing myfile.txt
Numbers found:
...
Enter file name:
invalidfile.txt
Opening invalidfile.txt
Could not find
invalidfile.txt
Closing invalidfile.txt
Numbers found:
```

```
nums = []
rd_nums = -1
my_file = input('Enter file name: ')

try:
    print('Opening', my_file)
    rd_nums = open(my_file, 'r')  # Might
cause IOError

    for line in rd_nums:
        nums.append(int(line))  # Might
cause ValueError
except IOError:
    print('Could not find', my_file)
except ValueError:
    print('Could not read number from',
my_file)
finally:
    print('Closing', my_file)
    if rd_nums != -1:
        rd_nums.close()
    print('Numbers found:', ' '.join([str(n)
for n in nums]))
```


PARTICIPATION ACTIVITY 10.5.2: Finally.

Assume that the following function has been defined.

```
def divide(a, b):
    z = -1
    try:
        z = a / b
    except ZeroDivisionError:
        print('Cannot divide by zero')
    finally:
        print('Result is', z)
```

1) What is the output of divide(4, 2)?

- Cannot divide by zero. Result is -1.
- Cannot divide by zero. Result is 2.0.
- Result is 2.0.

2) What is the output of divide(4, 0)?

Cannot divide by zero.
Result is -1.

- Cannot divide by zero.
  Result is 2.0.
- Result is 0.0.

10.6 Custom exception types

When raising an exception, a programmer can use the existing built-in exception types. For example, if an exception should be raised when the value of my_num is less than 0, the programmer might use a ValueError, as in `raise ValueError("my_num < 0")`. Alternatively, a ***custom exception type*** can be defined and then raised. The following example shows how a custom exception type LessThanZeroError might be used.

Figure 10.6.1: Custom exception types.

```
# Define a custom exception type
class LessThanZeroError(Exception):
    def __init__(self, value):
        self.value = value

my_num = int(input('Enter number: '))

if my_num < 0:
    raise LessThanZeroError('my_num
must be greater than 0')
else:
    print('my_num:', my_num)
```

```
Enter number: -100
Traceback (most recent call last):
  File "test.py", line 11, in
<module>
    raise LessThanZeroError('my_num
must be greater than 0')
__main__.LessThanZeroError
```


A programmer creates a custom exception type by creating a class that inherits from the built-in Exception class. The new class can contain a constructor, as shown above, that may accept an argument to be saved as an attribute. Alternatively, the class could have no constructor (and a "pass" statement might be used, since a class definition requires at least one statement). A custom exception class is typically kept bare,

adding a minimal amount of functionality to keep track of information that an exception handler might need. Inheritance is discussed in detail elsewhere.

Good practice is to include "Error" at the end of a custom exception type's name, as in LessThanZeroError or MyError. Custom exception types are useful to track and handle the unique exceptions that might occur in a program's code. Many larger third-party and Python standard library modules use custom exception types.

PARTICIPATION ACTIVITY 10.6.1: Custom exception types.

1) A custom exception type is usually defined by inheriting from the Exception class.
   - True
   - False

2) The following statement defines a new type of exception: `def MyMultError: pass`
   - True
   - False

3) "FileNotOpen" is a good name for a custom exception class.
   - True
   - False

10.7 LAB: Fat-burning heart rate

Write a program that calculates an adult's fat-burning heart rate, which is 70% of 220 minus the person's age. Complete fat_burning_heart_rate() to calculate the fat burning heart rate.

The adult's age must be between the ages of 18 and 75 inclusive. If the age entered is not in this range, raise a ValueError exception in get_age() with the message "Invalid age." Handle the exception in __main__ and print the ValueError message along with "Could not calculate heart rate info."

Ex: If the input is:

```
35
```


the output is:

```
Fat burning heart rate for a 35 year-old: 129.5 bpm
```

If the input is:

```
17
```

the output is:

```
Invalid age.
Could not calculate heart rate info.
```

LAB ACTIVITY 10.7.1: LAB: Fat-burning heart rate 0 / 10

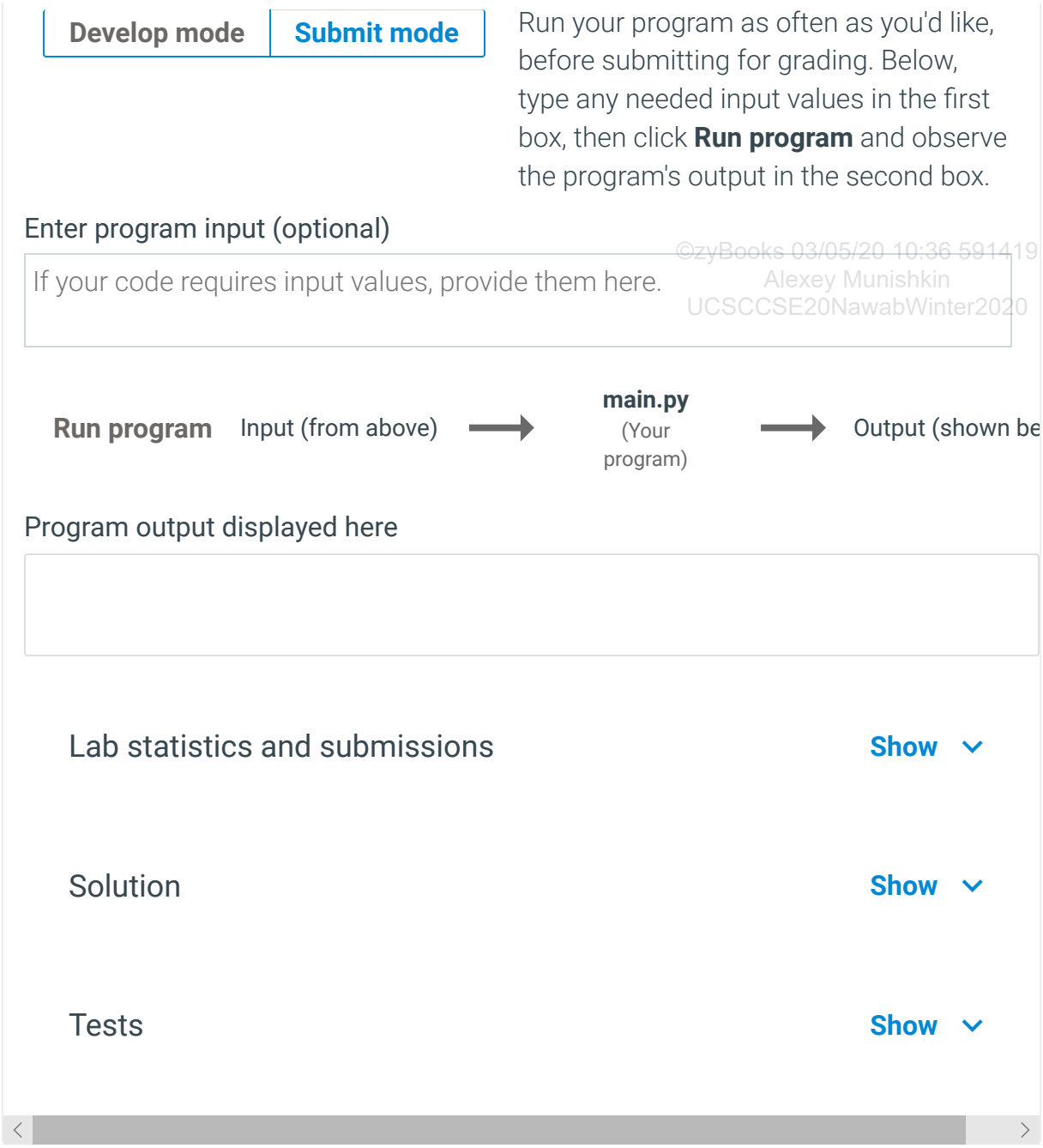
main.py Load default template...

```
def get_age():
    age = int(input())
    # TODO: Raise excpetion for invalid ages
    return age

# TODO: Complete fat_burning_heart_rate() function
def fat_burning_heart_rate(age):

    return heart_rate

if __name__ == "__main__":
    # TODO: Modify to call get_age() and fat_burning_heart_rate()
    #       and handle the exception
    age = get_age()
```



10.8 LAB: Exception handling to detect input string vs. integer

The given program reads a list of single-word first names and ages (ending with -1), and outputs that list with the age incremented. The program fails and throws an

exception if the second input on a line is a string rather than an integer. At FIXME in the code, add try and except blocks to catch the ValueError exception and output 0 for the age.

Ex: If the input is:

```
Lee 18
Lua 21
Mary Beth 19
Stu 33
-1
```


then the output is:

```
Lee 19
Lua 22
Mary 0
Stu 34
```

LAB ACTIVITY 10.8.1: LAB: Exception handling to detect input string vs. integer 0 / 10

main.py Load default template...

```
1  # Split input into 2 parts: name and age
2  parts = input().split()
3  name = parts[0]
4  while name != '-1':
5      # FIXME: The following line will throw ValueError exception.
6      #        Insert try/except blocks to catch the exception.
7      age = int(parts[1]) + 1
8      print('{} {}'.format(name, age))
9
10     # Get next line
11     parts = input().split()
12     name = parts[0]
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → Output (shown be

Program output displayed here

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

©zyBooks 03/05/20 10:36 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020