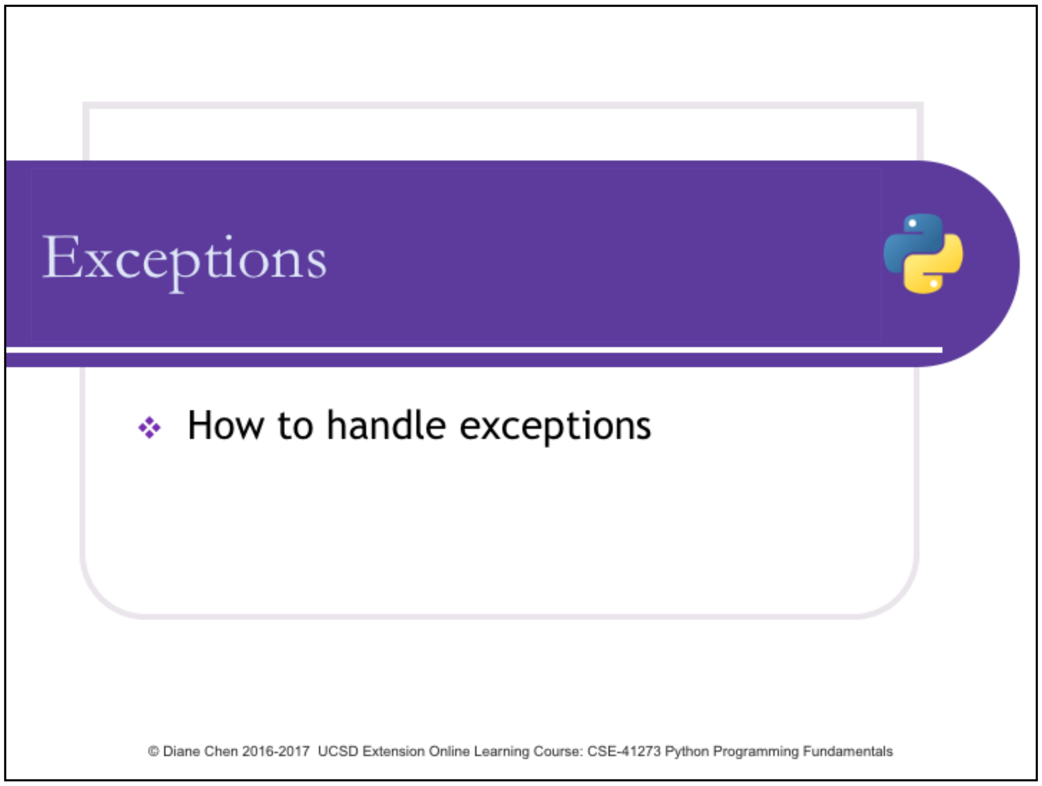
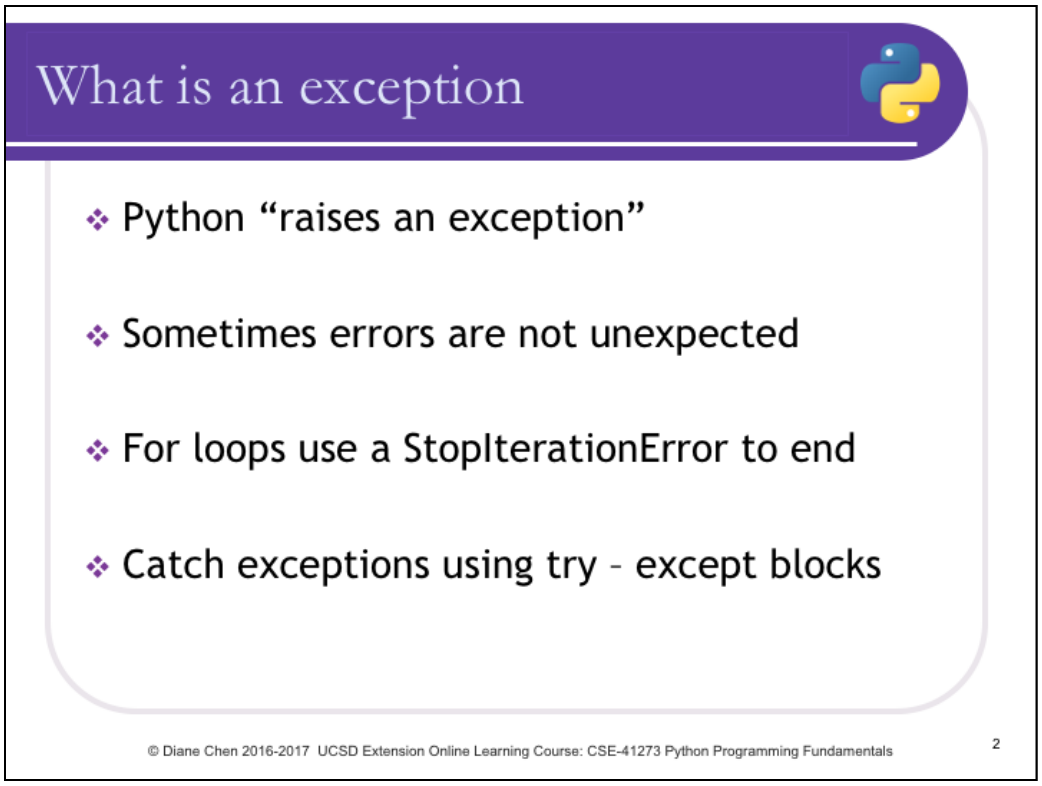


Exceptions

- How to handle exceptions

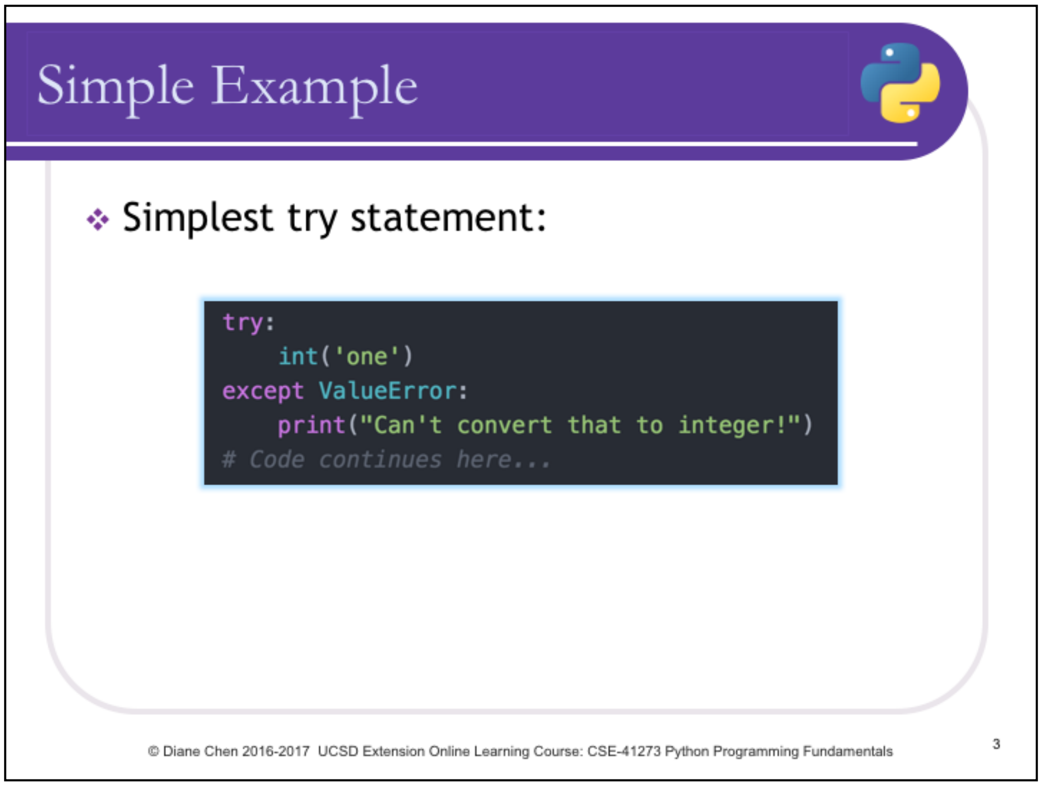
This lesson will cover exceptions; what they are and how to handle them.



What is an exception

- Python “raises an exception”
- Sometimes errors are not unexpected
- For loops use a StopIterationError to end
- Catch exceptions using try – except blocks

Exceptions happen when there is a problem in the code that Python doesn’t know how to deal with. It could be anything that disrupts the normal flow of execution of the program. We say that Python “raises an exception” when this happens. There are many types of exceptions, depending upon the type of error that occurs. Handling exceptions is an important part of programming. Not all exceptions are unexpected or disastrous. As an example, when we are iterating over something in a for loop, when the iterable is exhausted, an exception called a StopIterationError is raised. The code that controls for loops internally catches the exception and ends the for loop gracefully. When we want to catch exceptions, we use try – except blocks in our code. I have used the term "catch an exception". You will also hear people say "trap an exception" - the meaning is the same in this case.



Simple Example

- Simplest try statement:

```python
try:
    int('one')
except ValueError:
    print("Can't convert that to integer!")
# Code continues here...
```

The simplest form of try-except block is shown here. Inside the try block is code that might cause an exception. Inside the except block is code to handle the exception.

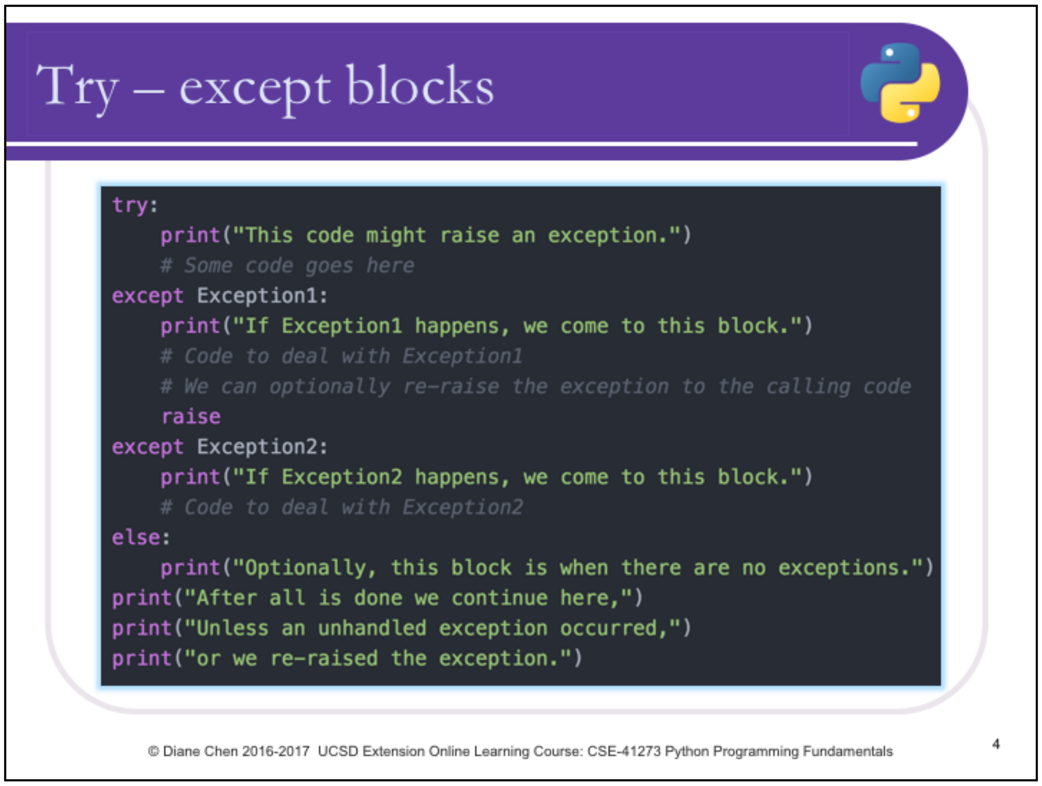
I will explain how the try statement works.

First, the try block (the statement(s) between the try and except keywords) is executed. This is the statement to convert the string to an int in my example.

If no exception occurs, the except block is skipped and execution of the try statement is finished. Code execution continues after the except block.

If an exception occurs during execution of the try block, the rest of the try block is skipped. In this case, there is nothing else in the try block other than the conversion of int. If the exception's type matches the exception given after the except keyword, the except block is executed. In our case a ValueError is raised, which matches the exception we want to catch, so it would print "Can't convert that to integer!"

If an exception occurs which does not match the exception named in the except statement, it is passed on to any enclosing try statements to handle the exception, which may be anywhere in the calling chain. If no try handler is found, it is an unhandled exception and program execution stops and prints the error information.



Try – except blocks

```
try:
    print("This code might raise an exception.")
    # Some code goes here
except Exception1:
    print("If Exception1 happens, we come to this block.")
    # Code to deal with Exception1
    # We can optionally re-raise the exception to the calling code
    raise
except Exception2:
    print("If Exception2 happens, we come to this block.")
    # Code to deal with Exception2
else:
    print("Optionally, this block is when there are no exceptions.")
print("After all is done we continue here,")
print("Unless an unhandled exception occurred,")
print("or we re-raised the exception.")
```

4

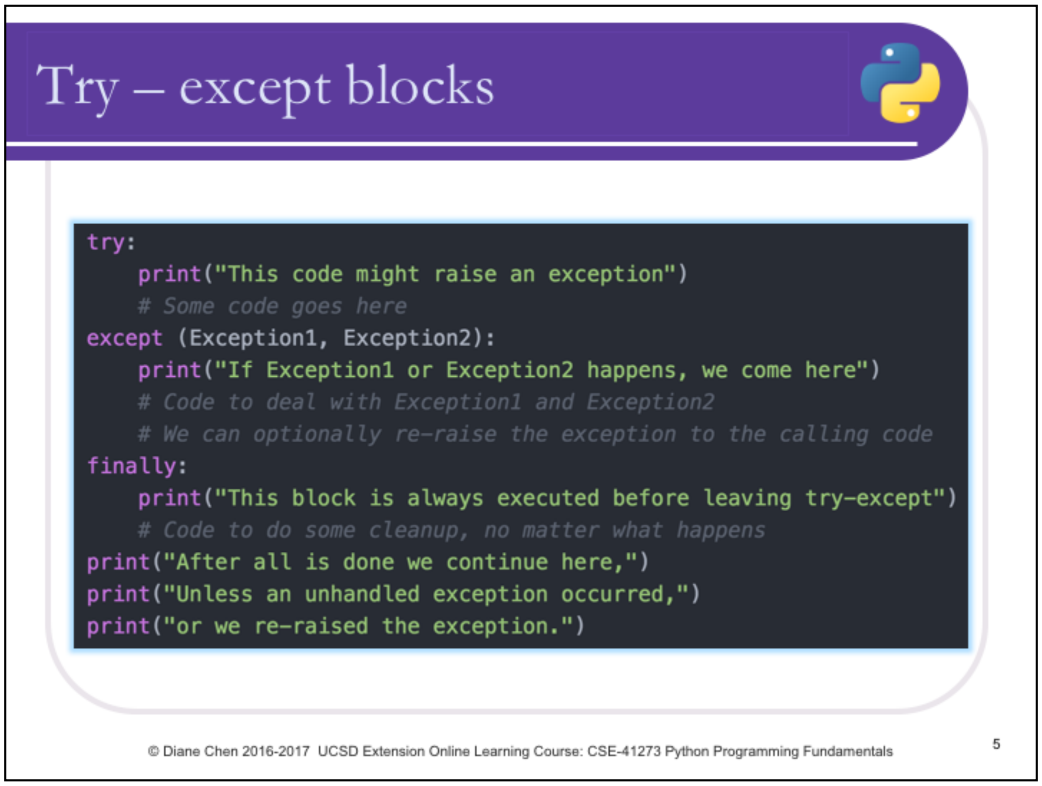
Here is one basic structure of try-except blocks. We can have multiple except blocks for different exceptions. Inside the except blocks is code to deal with the exception. Sometimes it is not fatal, we just need to do something like write to a log file or print a message to the user, or something like that.

If we don't want to continue, we can re-raise the error with a bare raise statement, which will raise the error to the closest enclosing try block – remember, we might be well down into a long calling chain, and some code that called us might have enclosed the call in a try block.

If we don't re-raise the error, then execution continues after the try-except blocks.

The else block is optional; the code in it will only be executed if no exception occurred. In other words, if we do something in an except block that handles the error and we just want to continue, this else block will be skipped.

If we did not re-raise the error in an except block, execution continues after the else block.

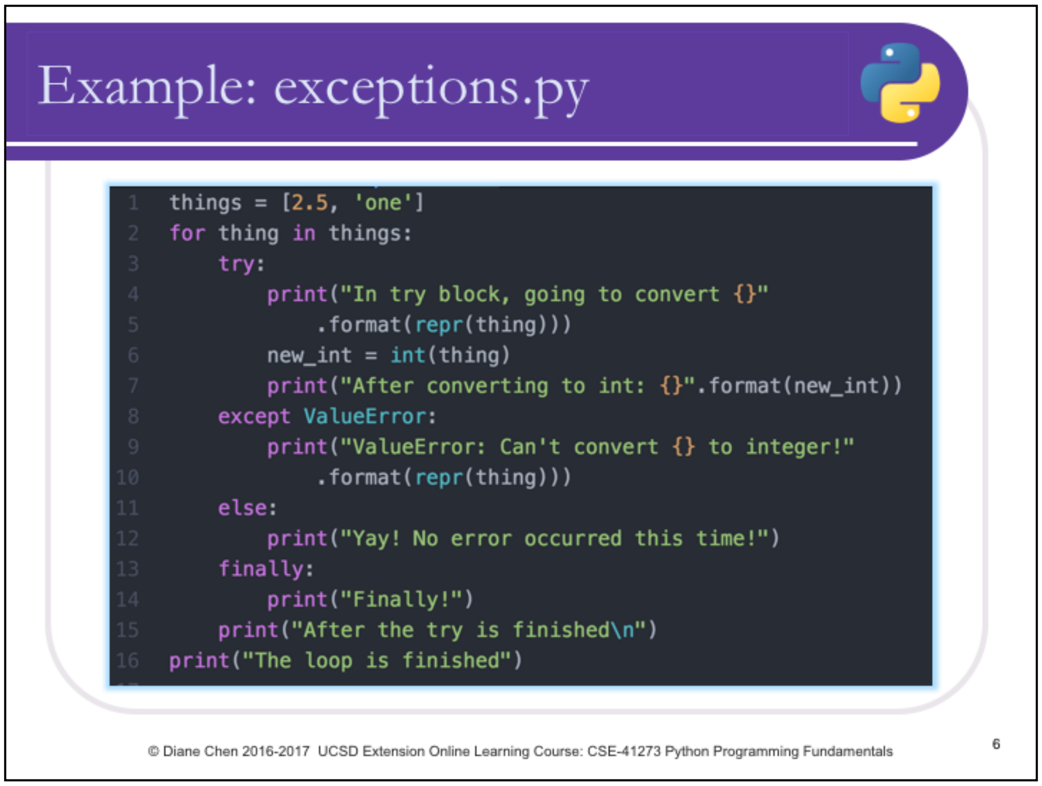


Try – except blocks

```python
try:
    print("This code might raise an exception")
    # Some code goes here
except (Exception1, Exception2):
    print("If Exception1 or Exception2 happens, we come here")
    # Code to deal with Exception1 and Exception2
    # We can optionally re-raise the exception to the calling code
finally:
    print("This block is always executed before leaving try-except")
    # Code to do some cleanup, no matter what happens
print("After all is done we continue here,")
print("Unless an unhandled exception occurred,")
print("or we re-raised the exception.")
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 5

Here is another example. We can handle more than one exception in one except block by enclosing them in parentheses to make them a tuple. The ”finally” block is a special block: it will always be executed no matter what happens. If no error occurred, the finally block is executed when the try block is finished. If an error occurs that we handle, the finally block is executed before we continue after the try –except block. If an unhandled exception occurs or we re-raise the exception, the finally block is executed before control is passed up the call chain.



Example: exceptions.py

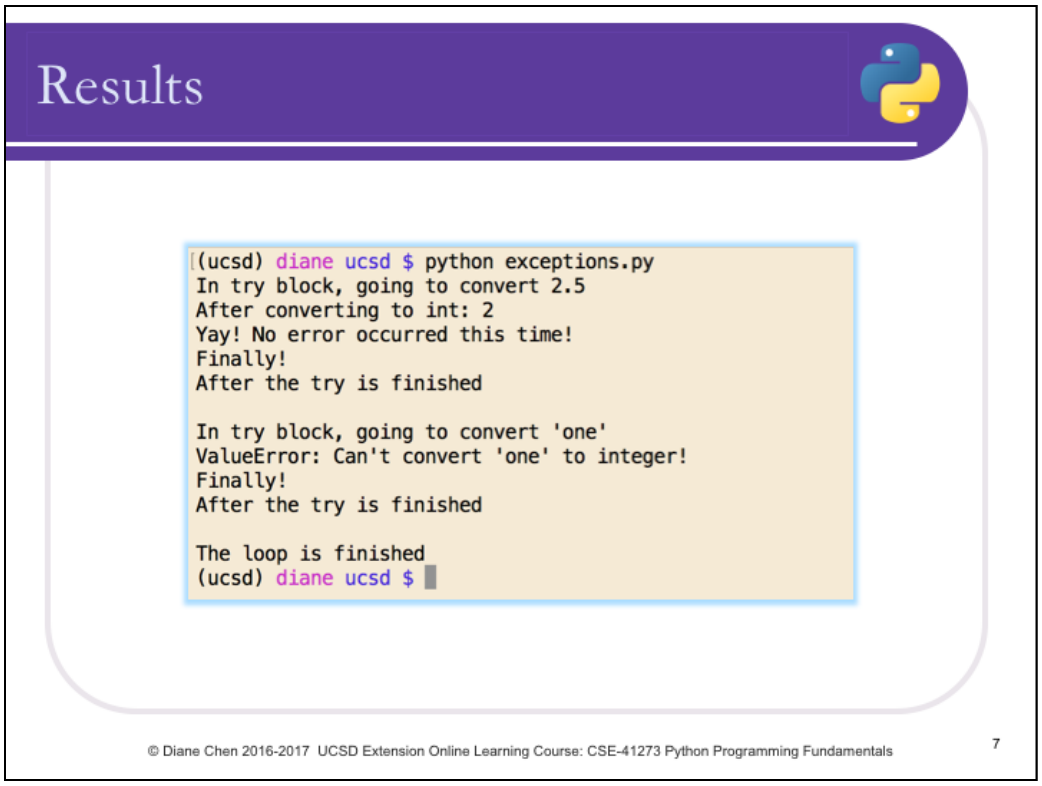
```python
things = [2.5, 'one']
for thing in things:
    try:
        print("In try block, going to convert {}"
            .format(repr(thing)))
        new_int = int(thing)
        print("After converting to int: {}".format(new_int))
    except ValueError:
        print("ValueError: Can't convert {} to integer!"
            .format(repr(thing)))
    else:
        print("Yay! No error occurred this time!")
    finally:
        print("Finally!")
    print("After the try is finished\n")
print("The loop is finished")
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 6

To give you an idea of how the try-except works, here is a little program to show the execution flow of a try-except block. I start with a list “things” containing two items: a float of 2.5, and a string ‘one’.

Then there is a for loop that loops over each item in things. After that, the the try-except block starts. Inside the try block, I print the item to be converted. Note I use the repr() function; this is so I will get an idea of the variable’s type. Next I attempt the conversion to an int. The next statement, still inside the try block, prints the result of the conversion.

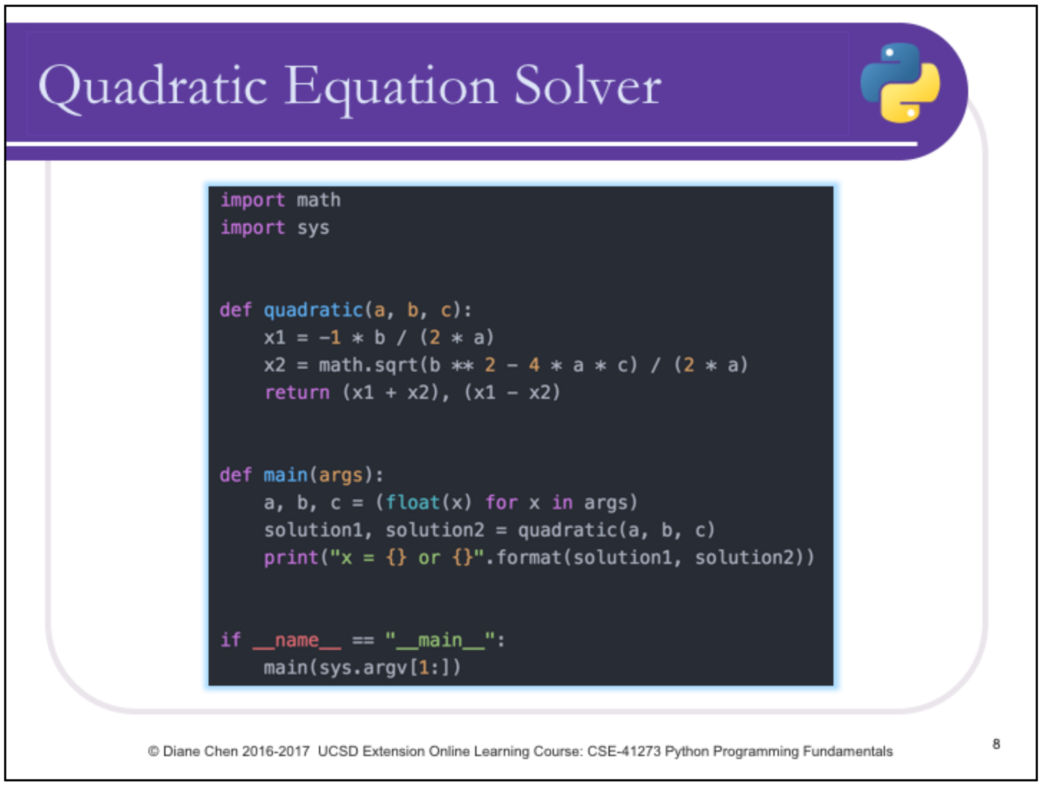
I have one except block to catch a ValueError exception, and print an error message. The else and finally blocks each print messages too. Then, still inside the for loop, but outside the try-except block, it prints a message and after the loop completes, it prints another message.



Results

```
(ucsd) diane ucsd $ python exceptions.py
In try block, going to convert 2.5
After converting to int: 2
Yay! No error occurred this time!
Finally!
After the try is finished

In try block, going to convert 'one'
ValueError: Can't convert 'one' to integer!
Finally!
After the try is finished

The loop is finished
(ucsd) diane ucsd $
```

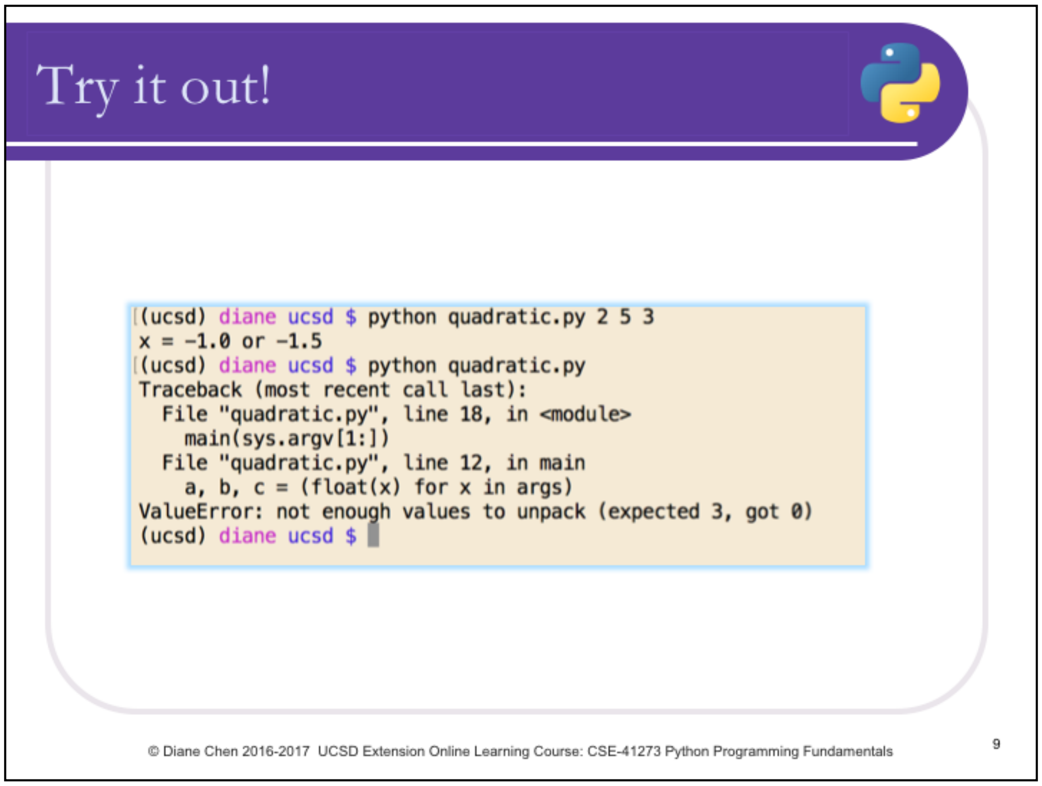
This is the output when it is run. First it prints the message it is in the try block and is going to convert 2.5, then it successfully converts it and prints the result. Because no error occurred, the else block prints ”Yay! No error occurred this time!”. Now it is done, so it prints “Finally!” from the finally block, then the message after the try is finished. Now it goes through the loop a second time. This time you can see that it goes into the try block and prints that it is going to convert the string ‘one’. We don’t get a message from after the conversion because a ValueError occurred & it goes to the except block and prints the message that it can’t convert the string ‘one’ to integer. Then it goes to the finally block and prints that message, then the messages from after the try is finished and then the message that the loop is finished.



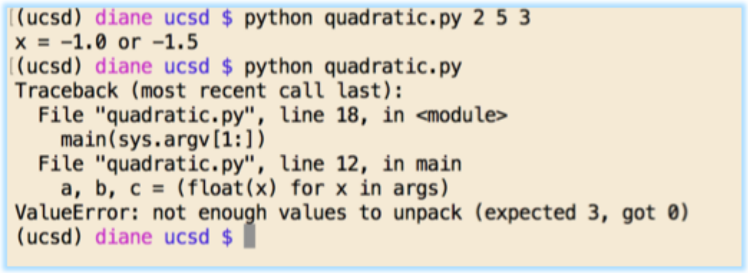
Quadratic Equation Solver

```
import math
import sys


def quadratic(a, b, c):
    x1 = -1 * b / (2 * a)
    x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
    return (x1 + x2), (x1 - x2)


def main(args):
    a, b, c = (float(x) for x in args)
    solution1, solution2 = quadratic(a, b, c)
    print("x = {} or {}".format(solution1, solution2))


if __name__ == "__main__":
    main(sys.argv[1:])
```

Let's go through an example that is a little bit more like a real world project. Here is a program that solves quadratic equations. You don't have to know quadratic equations – that part is already in the program!

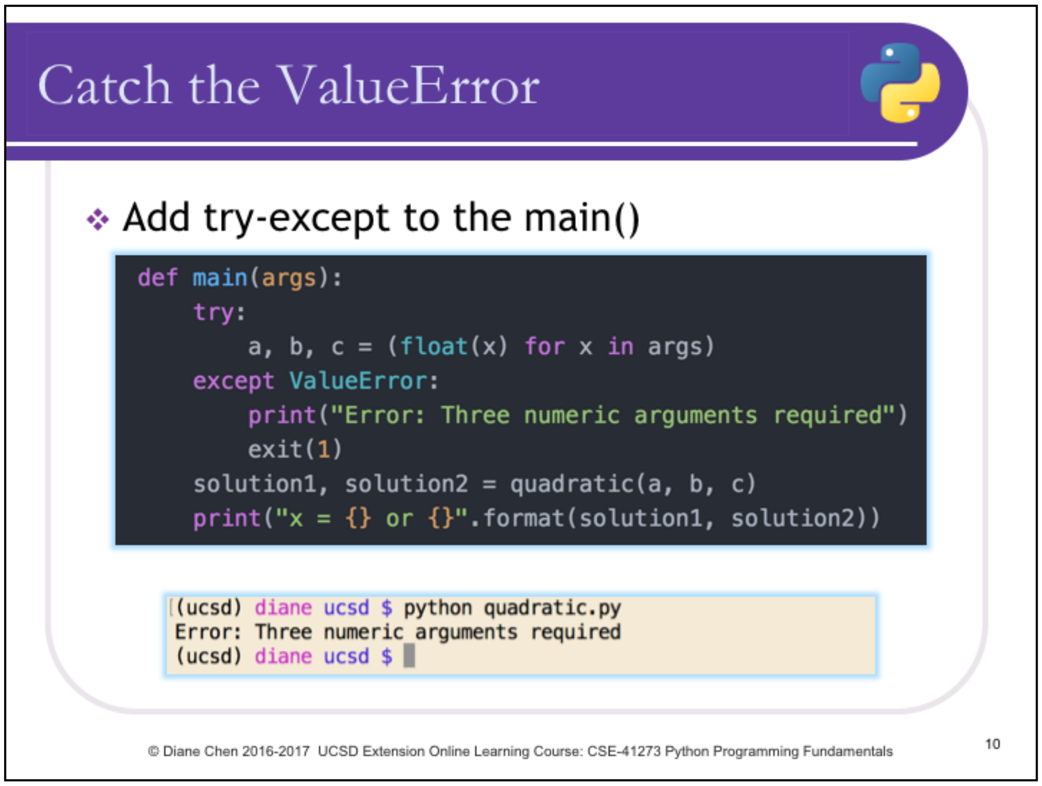


Try it out!



```
[(ucsd) diane ucsd $ python quadratic.py 2 5 3
x = -1.0 or -1.5
[(ucsd) diane ucsd $ python quadratic.py
Traceback (most recent call last):
  File "quadratic.py", line 18, in <module>
    main(sys.argv[1:])
  File "quadratic.py", line 12, in main
    a, b, c = (float(x) for x in args)
ValueError: not enough values to unpack (expected 3, got 0)
(ucsd) diane ucsd $ 
```

Let’s try it out. When we run it with the values 2, 5, 3, we get an answer. What if we call it with no arguments? We get an error because it was expecting 3 values. Because we’re programmers, we know what went wrong, but it’s not very user-friendly for other users of our program. They aren’t going to know what that means. Let’s do something about that.



Catch the ValueError

- Add try-except to the main()

```
def main(args):
    try:
        a, b, c = (float(x) for x in args)
    except ValueError:
        print("Error: Three numeric arguments required")
        exit(1)
    solution1, solution2 = quadratic(a, b, c)
    print("x = {} or {}".format(solution1, solution2))
```

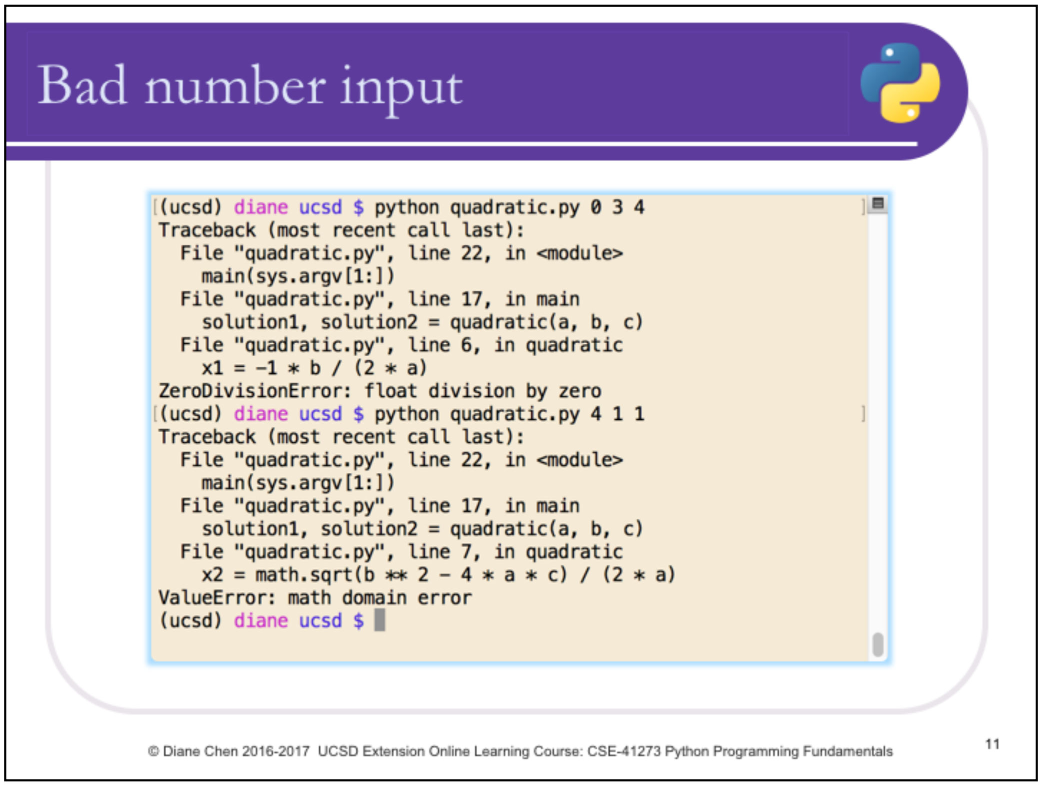
```
(ucsd) diane ucsd $ python quadratic.py
Error: Three numeric arguments required
(ucsd) diane ucsd $ 
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals

10

Here we can fix that error. We'll put the unpacking statement into a try-except block and check for ValueError. If we get a ValueError, we print an informative message to the user.

We don't want to continue after we handle the exception, so after we print the message, we call the system exit() function to exit Python. exit(0) means no error and any other value (we arbitrarily use 1) means there was an error. By default when a program ends normally, it is as if exit(0) was called. And here we see that if we call the program with no arguments we get a more sensible error message.

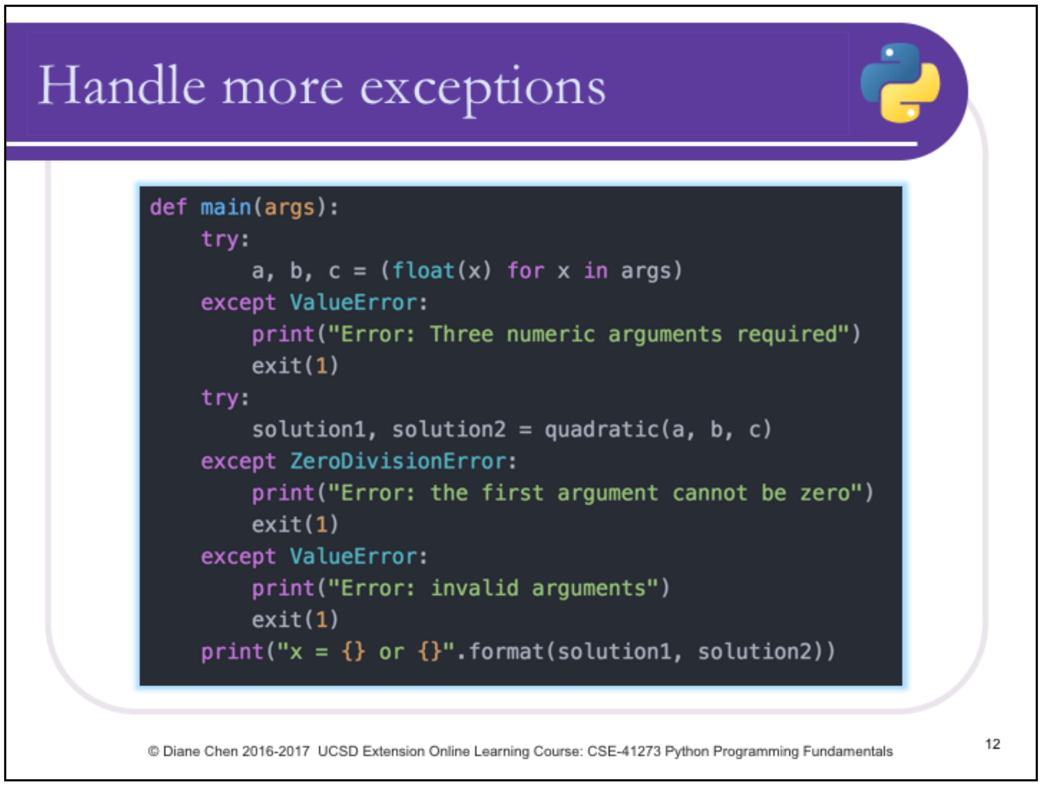


Bad number input

```
(ucsd) diane ucsd $ python quadratic.py 0 3 4
Traceback (most recent call last):
  File "quadratic.py", line 22, in <module>
    main(sys.argv[1:])
  File "quadratic.py", line 17, in main
    solution1, solution2 = quadratic(a, b, c)
  File "quadratic.py", line 6, in quadratic
    x1 = -1 * b / (2 * a)
ZeroDivisionError: float division by zero
(ucsd) diane ucsd $ python quadratic.py 4 1 1
Traceback (most recent call last):
  File "quadratic.py", line 22, in <module>
    main(sys.argv[1:])
  File "quadratic.py", line 17, in main
    solution1, solution2 = quadratic(a, b, c)
  File "quadratic.py", line 7, in quadratic
    x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
ValueError: math domain error
(ucsd) diane ucsd $
```

11

If we input a zero for the first argument, we get an error, because that is an invalid number for a quadratic equation – trust me on that. It's another unhandled exception, this time a ZeroDivisionError from the quadratic function. Also, if we enter the numbers so that it tries to take a square root of a negative number, we get a different kind of ValueError, this time a math domain error. Neither of those are particularly helpful to a user.

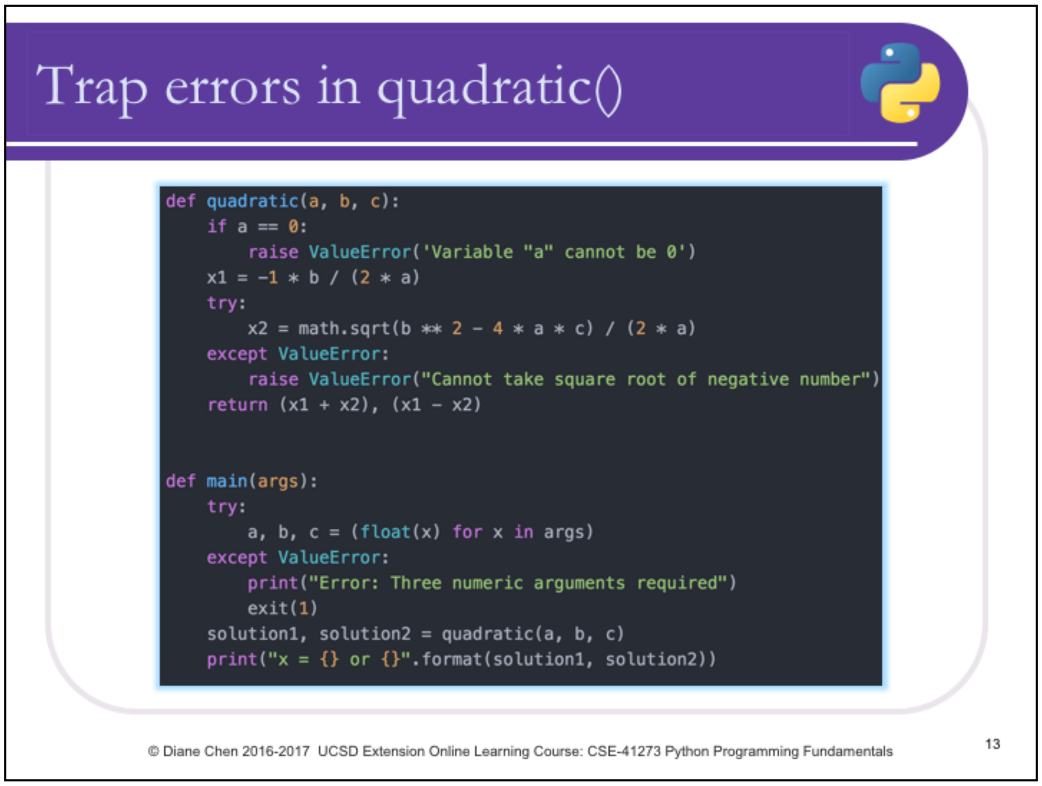


Handle more exceptions

```python
def main(args):
    try:
        a, b, c = (float(x) for x in args)
    except ValueError:
        print("Error: Three numeric arguments required")
        exit(1)
    try:
        solution1, solution2 = quadratic(a, b, c)
    except ZeroDivisionError:
        print("Error: the first argument cannot be zero")
        exit(1)
    except ValueError:
        print("Error: invalid arguments")
        exit(1)
    print("x = {} or {}".format(solution1, solution2))
```

12

Let’s change the main program again to catch both of those errors. We add a new try-except set with two except blocks to the program, one to catch the zero division error, and one to catch the value error. Always having to call exit() each time is tiresome and perhaps we should catch these exceptions in the quadratic function itself. In that way, someone using our quadratic function from the module will get nicer messages too.



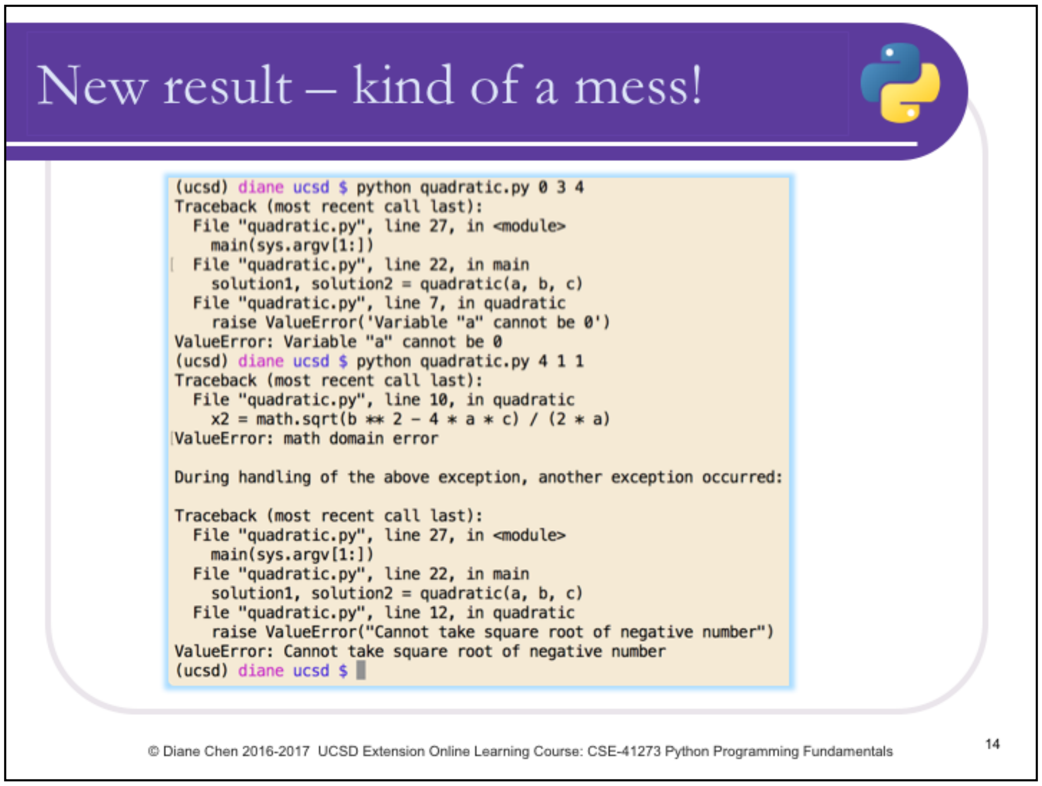
Trap errors in quadratic()

```python
def quadratic(a, b, c):
    if a == 0:
        raise ValueError('Variable "a" cannot be 0')
    x1 = -1 * b / (2 * a)
    try:
        x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
    except ValueError:
        raise ValueError("Cannot take square root of negative number")
    return (x1 + x2), (x1 - x2)


def main(args):
    try:
        a, b, c = (float(x) for x in args)
    except ValueError:
        print("Error: Three numeric arguments required")
        exit(1)
    solution1, solution2 = quadratic(a, b, c)
    print("x = {} or {}".format(solution1, solution2))
```

13

In the quadratic function, we check for the ”a” coefficient being zero, and raise our own ValueError for that. Then if we get the ValueError on the calculations, we raise the ValueError with a more informative error message.

Note that we do not exit the program from the quadratic function. We could do that, but it is not a good programming practice, because someone could be importing the quadratic module to use our quadratic function. The user of our function can put it into a try-except block to catch the errors generated, but cannot protect against the function exiting. We always want to raise exceptions from functions like this so that the caller of the function can choose to catch errors or not.



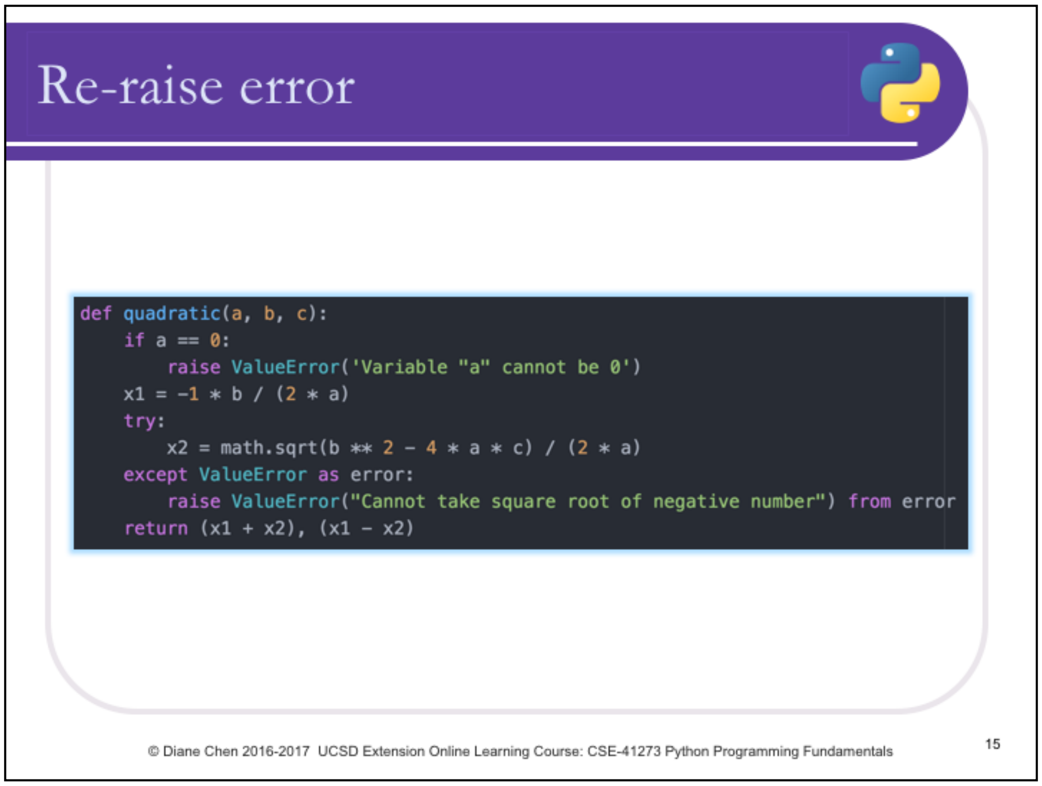
New result – kind of a mess!

```
(ucsd) diane ucsd $ python quadratic.py 0 3 4
Traceback (most recent call last):
  File "quadratic.py", line 27, in <module>
    main(sys.argv[1:])
  File "quadratic.py", line 22, in main
    solution1, solution2 = quadratic(a, b, c)
  File "quadratic.py", line 7, in quadratic
    raise ValueError('Variable "a" cannot be 0')
ValueError: Variable "a" cannot be 0
(ucsd) diane ucsd $ python quadratic.py 4 1 1
Traceback (most recent call last):
  File "quadratic.py", line 10, in quadratic
    x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
ValueError: math domain error

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "quadratic.py", line 27, in <module>
    main(sys.argv[1:])
  File "quadratic.py", line 22, in main
    solution1, solution2 = quadratic(a, b, c)
  File "quadratic.py", line 12, in quadratic
    raise ValueError("Cannot take square root of negative number")
ValueError: Cannot take square root of negative number
(ucsd) diane ucsd $
```

In the first one with the zero coefficient, we still get the traceback, but the actual error message makes more sense because it specifically tells us that we entered an invalid value for the “a” coefficient.

The second one, though, is rather confusing. It says we got an error while handling another error. This is exactly what happened because we didn’t like the “math domain error” message & wanted our own,so we raised the ValueError with a better message.

But the output is confusing and we don’t like that.

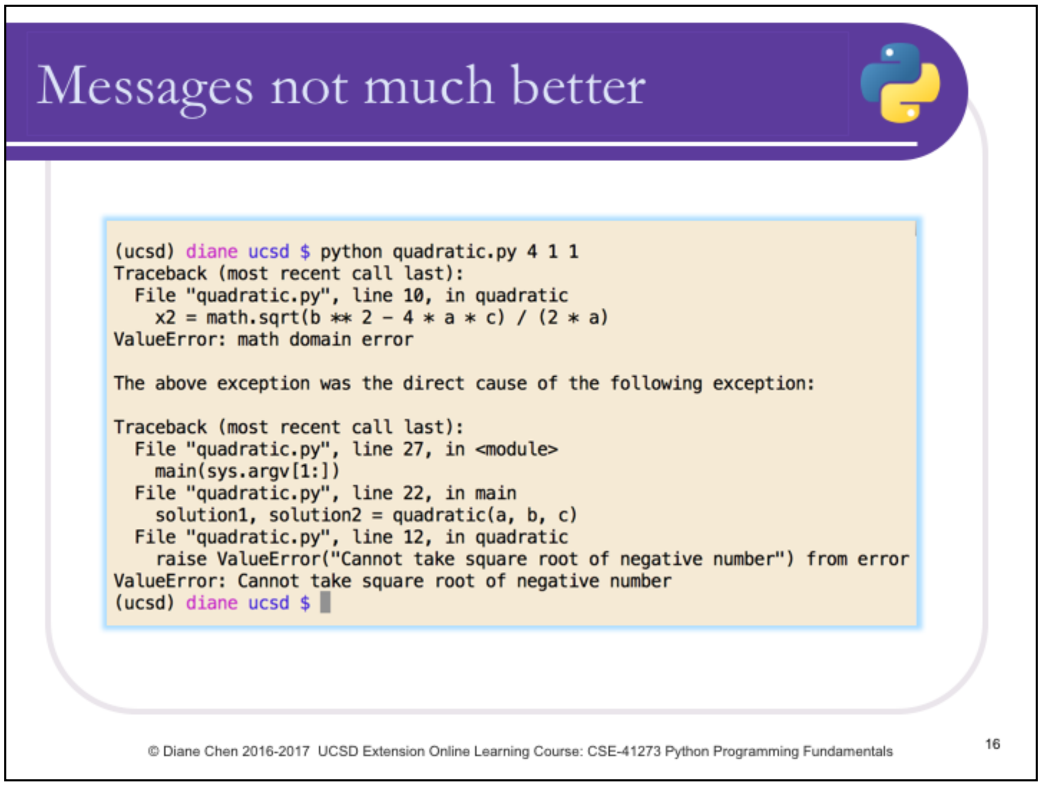


Re-raise error

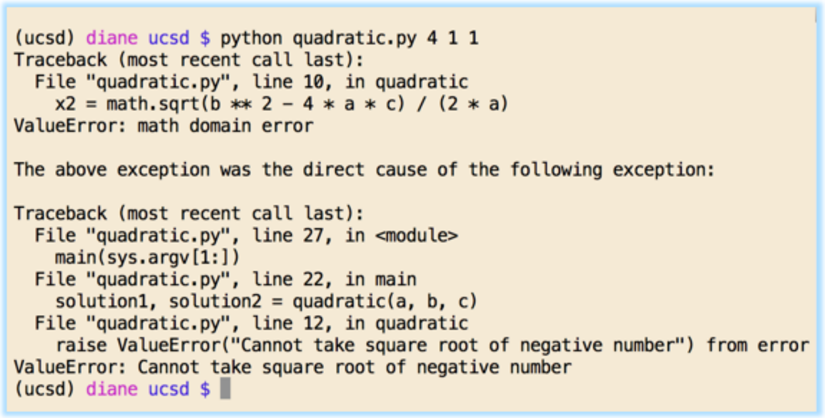
```python
def quadratic(a, b, c):
    if a == 0:
        raise ValueError('Variable "a" cannot be 0')
    x1 = -1 * b / (2 * a)
    try:
        x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
    except ValueError as error:
        raise ValueError("Cannot take square root of negative number") from error
    return (x1 + x2), (x1 - x2)
```

15

Now we change the except statement to add “as error”. This saves the error instance in the variable error. Then we raise the new ValueError “from error”, so the system knows it’s the same error.



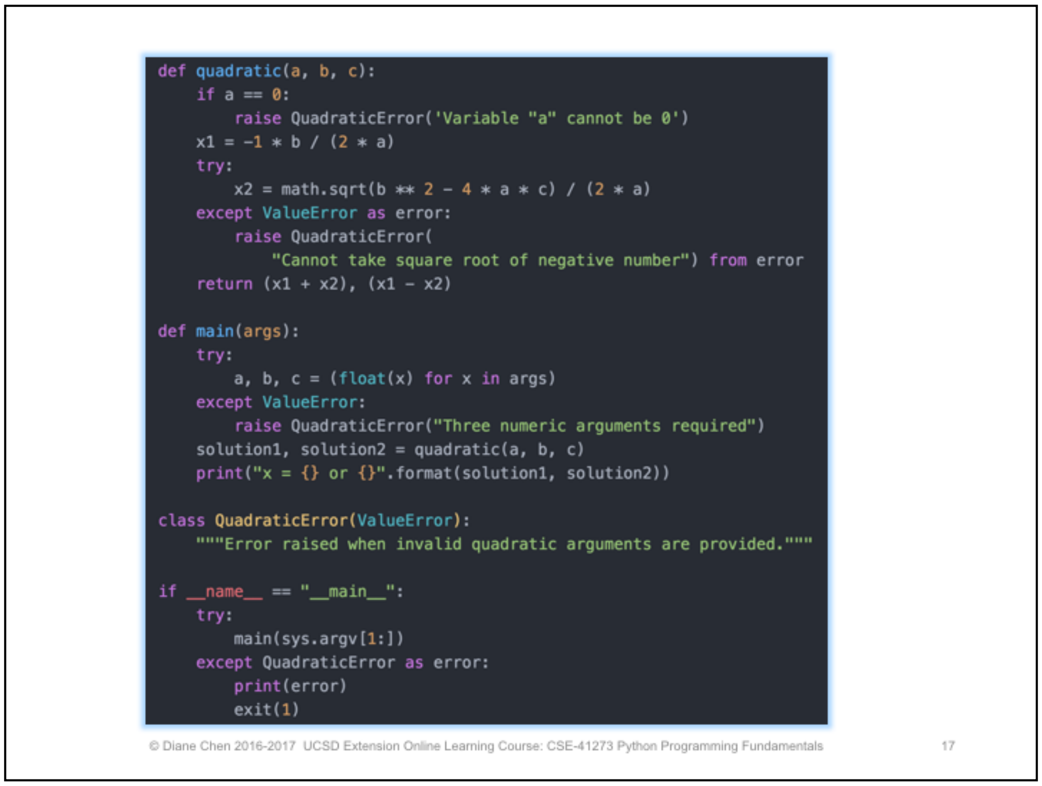
Messages not much better



```
(ucsd) diane ucsd $ python quadratic.py 4 1 1
Traceback (most recent call last):
  File "quadratic.py", line 10, in quadratic
    x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
ValueError: math domain error

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "quadratic.py", line 27, in <module>
    main(sys.argv[1:])
  File "quadratic.py", line 22, in main
    solution1, solution2 = quadratic(a, b, c)
  File "quadratic.py", line 12, in quadratic
    raise ValueError("Cannot take square root of negative number") from error
ValueError: Cannot take square root of negative number
(ucsd) diane ucsd $ 
```

16

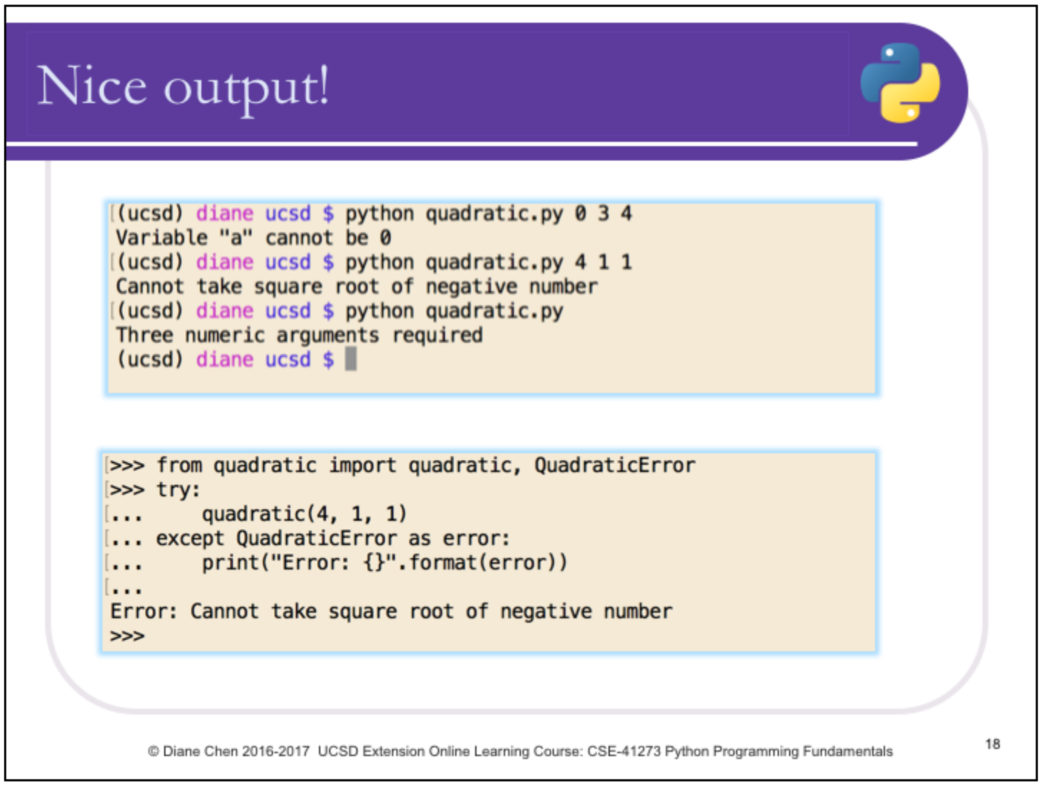
Here Python knows that the error that we raised came as a result of the first math domain error. But really, although this makes more sense to a developer, it’s not really an improvement to a user, is it? Users never want to see tracebacks. So let's see what we can do about that!



```python
def quadratic(a, b, c):
    if a == 0:
        raise QuadraticError('Variable "a" cannot be 0')
    x1 = -1 * b / (2 * a)
    try:
        x2 = math.sqrt(b ** 2 - 4 * a * c) / (2 * a)
    except ValueError as error:
        raise QuadraticError(
            "Cannot take square root of negative number") from error
    return (x1 + x2), (x1 - x2)

def main(args):
    try:
        a, b, c = (float(x) for x in args)
    except ValueError:
        raise QuadraticError("Three numeric arguments required")
    solution1, solution2 = quadratic(a, b, c)
    print("x = {} or {}".format(solution1, solution2))

class QuadraticError(ValueError):
    """Error raised when invalid quadratic arguments are provided."""

if __name__ == "__main__":
    try:
        main(sys.argv[1:])
    except QuadraticError as error:
        print(error)
        exit(1)
```

17

Let’s refactor our code to make it so we can print informative messages and exit the system nicely. We make our own QuadraticError that inherits from the ValueError class. I know we haven’t talked about classes or inheritance yet, but bear with me. What we’re doing here is simply creating a new error type that is like a ValueError. In the quadratic function, we raise a QuadraticError for each of the problems we detect. In the main function, we also raise a QuadraticError when there are not 3 arguments, rather than handling the error there. Then, in our block that gets executed when we are called from the command line, we add a new try-except block to catch a QuadraticError if it occurs. Then we print the error message and exit.

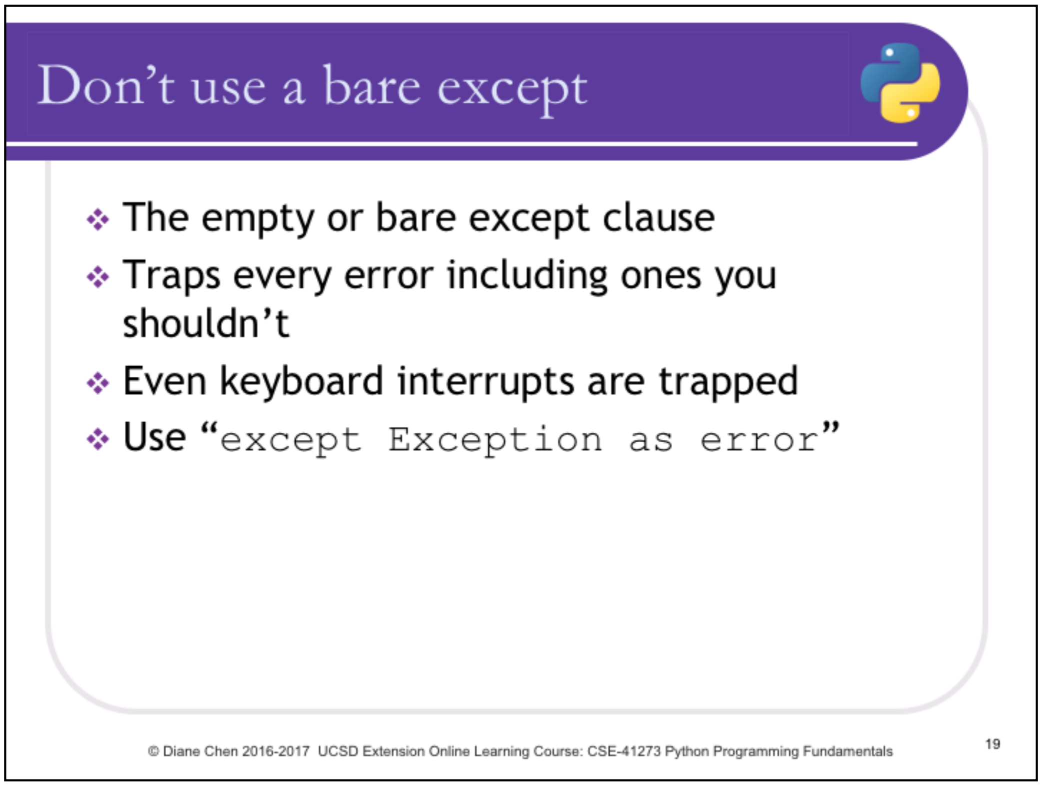


Nice output!

```
(ucsd) diane ucsd $ python quadratic.py 0 3 4
Variable "a" cannot be 0
(ucsd) diane ucsd $ python quadratic.py 4 1 1
Cannot take square root of negative number
(ucsd) diane ucsd $ python quadratic.py
Three numeric arguments required
(ucsd) diane ucsd $ 
```

```
>>> from quadratic import quadratic, QuadraticError
>>> try:
...     quadratic(4, 1, 1)
... except QuadraticError as error:
...     print("Error: {}".format(error))
...
Error: Cannot take square root of negative number
>>>
```

Now we get nice output from each error that we've found. And if we want to use the Quadratic function in our module from somewhere else, we can trap specifically for the QuadraticError and still print a nice error message. That was all a little roundabout to get to this point, but now you should understand how exceptions work, and will be able to use them when needed.



Don’t use a bare except

- The empty or bare except clause
- Traps every error including ones you shouldn’t
- Even keyboard interrupts are trapped
- Use “`except Exception as error`”

Sometimes people think “Why don’t I just trap every error?” When you have a try-except block where no error type is specified in the except clause, all errors will get trapped.

But this is not a good thing to do, because even system-exiting errors such as a keyboard interrupt are trapped, so you can't stop the program with control-C like you normally could. And, since there are many, many different error types, once you get to the except clause, how do you know you will be able to handle it properly?

If you feel you must trap all errors, use the exception type Exception with a capital E. If you make your except clause “except Exception as error”, then you will trap all non-system-exiting errors and you will have the error information in the variable error. From there you can handle the exceptions as you wish.

However, if you ever feel the need to catch all exceptions like this, you might want to re-think your program and consider ways to refactor it so you don’t need to do it. In many cases, probably most cases, you should just let the errors that you don't specifically know how to handle propagate up to a higher level and be handled elsewhere.