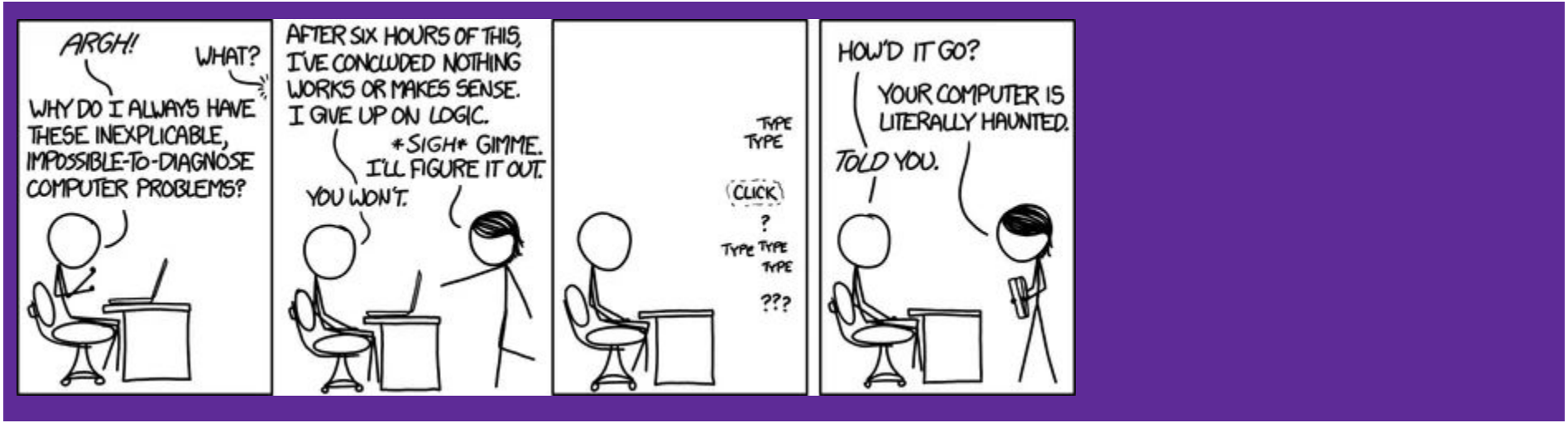
CS 1: Intro to CS

Intro to Error-Handling



Summary

Handling errors with Exceptions:

- Using **`raise`** to raise exceptions on invalid arguments
- Using **`try`**/**`except`** to handle raised exceptions

How to use:

- This slide deck is specific to error-handling (supplemented with error-handling-starter.py and Reading 19) for reference; we will prioritize raising errors with **`raise`** (used in OOP unit and MP6) and later **`try`**/**`except`**

Introducing Errors

Today, we will introduce error-handling in Python and:

1. Learn how to handle errors correctly
2. Understand better how Python works internally

First, let's motivate error-handling by thinking of possible errors we could account for in our programs.

Errors

Many possible errors can occur when running a Python program

What examples can you think of?

- Division by 0
- Accessing a non-existent index of a list
- Accessing a non-existent key in a dictionary
- Trying to store a new value into a tuple
- Trying to open a file (for reading) that does not exist
- ….

Errors

Consider:

```
>>> def print_reciprocal(x):
...     recip = 1.0 / x
...     print(f'reciprocal of {x} is {recip}')
>>> print_reciprocal(0)
```

This will not give a legal value (Python doesn't have infinitely large integers)

How should Python handle errors like this?

Handling Errors: 1st Attempt

Could ignore the error and just continue (“silent failure”)

What's a problem with this?

Not an effective way to proceed!

- We don't know *why* the error occurred or *where* it occurred

Handling Errors: 2nd Attempt

Could halt the program

Advantage: Prevents a crash

Problems?

- Too drastic (some errors can be recovered reasonably)
- Still don't know *why* error occurred or *where* it occurred

Handling Errors: 3rd Attempt

Could print an informative error message and halt the program

Advantages:

- Prevents a crash
- Know *why* error happened

Problems?

- Too drastic (some errors can be recovered reasonably)
- Still don't know *where* error occurred

Handling Errors: 4th Attempt

Could print an informative error message stating *why* and *where* the error occurred and halt the program

Advantages:

- Prevents a crash
- Know *why* and *where* error happened

Problems?

- Too drastic (some errors can be recovered reasonably)

This is what Python does by default

In Python

```
>>> a = 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

There are two components to this error message...

In Python

```
>>> a = 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The **error message** states what *kind* of error occurred and more specifically, *why* the error occurred in this particular case

In Python

```
>>> a = 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The **traceback** states *where* the error occurred

We'll look at this in more detail soon

Error Recovery

Errors do not have to result in the termination of the entire program

Many kinds of errors can be recovered from:

- Bad input -> prompt for new input
- Nonexistent files -> use a different file
- Invalid constructor argument -> use default value and assume client reads docstring on this default initialization.
- etc.

Error Recovery

No general rule on how best to recover from errors (specific to particular situation where error occurs)

We need a *general* mechanism

It should say:

- Under normal circumstances, do this
- If error A happens, do A'
- Else if error B happens, do B'
- etc. until all possibilities handled

Another Example

Suppose we want a function to compute roots of a quadratic equation (with two possible solutions for x returned as a tuple)

We want to solve this equation for x:

$a * x^2 + b * x + c = 0$

Solution 1: $(-b + sqrt(b^2 - 4*a*c)) / (2 * a)$

Solution 2: $(-b - sqrt(b^2 - 4*a*c)) / (2 * a)$

What is needed for these solutions to work?

Example

```
def solve_quadratic_equation(a, b, c):
    """
    Solves the quadratic equation
        a*x**2 + b*x + c == 0
    for x.
    """
    sol1 = (-b + sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
    sol2 = (-b - sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
    return (sol1, sol2)
```

Problem: What to do if **`a == 0`**?

This makes **`sol1`** and **`sol2`** uncomputable (division by 0)

Strategies

Could always use an **`if`** statement to check whether a was 0 before computing anything else

- Not a bad strategy, however usually **a** will not be 0
- Would rather have the code represent the typical case first and the exceptional cases only when the typical case fails
- Let's try a different approach

Introducing `try` and `except`

Python provides a special kind of statement for dealing with errors that can be recovered from: a **`try`/`except` statement**

We'll first show what this looks like in the context of our example, then explain it in detail

Note that if **`a == 0`**, the quadratic equation **`a*x**2 + b*x + c = 0`** becomes:

`b*x + c = 0`

or in terms of **`x`**,

`x = -c / b` (assuming **`b`** is nonzero)

Example using `try` and `except`

```
def solve_quadratic_equation(a, b, c):
    """
    Solves the quadratic equation
        a*x**2 + b*x + c == 0
    for x, returning a 2-element tuple of the solution.
    """
    try:
        sol1 = (-b + sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
        sol2 = (-b - sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
        return (sol1, sol2)
    except ZeroDivisionError: # a == 0
        sol = -c / b # assume b != 0
        return sol
```

`try` and `except`

Structure of a **`try`**/**`except`** statement:

```
try:
    <some code which may result in an error>
except <name of the error>:
    <code to run if the error occurs>
```

`try` and **`except`** say:

- Try to execute the block of code (**`try`** block)
- If a particular error occurs, execute this other block of code (**`except`** block)

`try` and `except`

Structure of a **`try`**/**`except`** statement:

```
try:
    <some code which may result in an error>
except <name of the error>:
    <code to run if the error occurs>
```

`try` and **`except`** are block-structured statements like **`if`**, **`for`**, and **`while`**

Can have multiple statements in a **`try`** or **`except`** block, but must all be indented the same

`try` and `except`: Rules

```
try:
    <some code which may result in an error>
except <name of the error>:
    <code to run if the error occurs>
```

You cannot use a **`try`** block without **`except`** (there's also no point if you could)

Also cannot have **`except`** block without a preceding **`try`** block

`try` and `except`: Rules

```
try:
    <some code which may result in an error>
except <error1>:
    <code to run if error1 occurs>
except <error2>:
    <code to run if error2 occurs>
```

Can also have multiple **`except`** blocks, each corresponding to a different kind of error

Rules: Nested `try` and `except`?

```
def solve_quadratic_equation(a, b, c):
    """Solves the quadratic equation a*x**2 + b*x + c == 0 for x."""
    try:
        sol1 = (-b + sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
        sol2 = (-b - sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
        return (sol1, sol2)
    except ZeroDivisionError: # a == 0
        try:
           sol = -c / b
           return sol
        except ZeroDivisionError: # b == 0
           print('No solution for x when a and b are 0')
```

Note: The nested **`try`**/**`except`** *could* be simplified with **`if`**/**`else`** in this case.

`try` and `except`: Rules

```
try:
    <some code which may result in an error>
except:
    <code to run if any error occurs>
```

Can also have a “catch-all” **`except`** block, which will execute if *any* kind of error occurs

Usually this is very poor design (not specific enough to the particular problem)

`try` and `except`: Rules

```
try:
    <some code which may result in an error>
except <error1>:
    <code to run if error1 occurs>
except <error2>:
    <code to run if error2 occurs>
except:
    <code to run if any other error occurs>
```

But OK sometimes to use catch-all exception handler *after* more specific handlers

More Rules

When evaluating a **`try`/`except`** statement, **only one `except` statement's code can be executed**

Even if there is a catch-all **`except`** block at the end, it's not executed if a previous except block's code was executed

Terminology

Errors that occur in code are called **exceptions**

- They represent “exceptional conditions”
- These don't *always* correspond to errors!

Signaling an error is called **raising an exception** (or **throwing an exception**)

Handling an error is called **catching an exception**

Therefore, exceptions are raised in **`try`** blocks and caught in **`except`** blocks

Exceptions

Exceptions are actual Python objects

They can have associated data

- Often, an error message that indicates more precisely what went wrong

Some examples of Python errors:

- **`ZeroDivisionError`** (division by zero)
- **`IndexError`** (list index out of bounds)
- **`FileNotFoundError`** (e.g. reading a non-existent file)
- And many, many others

try/except again

Let's look in detail at what happens when an exception is handled

Sample function:

```
def print_reciprocal(x):
    try:
        recip = 1.0 / x
        print('reciprocal of {} is {}'.format(x, recip))
    except ZeroDivisionError:
        print('Division by zero!')
```

try/except again

```
>>> print_reciprocal(5.0)

    try:
        recip = 1.0 / 5.0 # 0.2
        print('reciprocal of {} is {}'.format(5.0, 0.2))
    except ZeroDivisionError:
        print('Division by zero!')
```

This prints: **`reciprocal of 5 is 0.2`** (**except** block is never executed)

try/except again

```
>>> print_reciprocal(0.0)

    try: # 1.
        recip = 1.0 / 0.0  # 2. error is thrown here
        print('reciprocal of {} is {}'.format(x, recip))
    except ZeroDivisionError:  # 3.
        print('Division by zero!')  # 4.
```

This prints: **`Division by zero!`** (the second line in the **`try`** block is not executed, nothing else happens after the exception has been handled)

Another Example: What Lines are Executed?

```
def print_reciprocals():
    values = [-1, 0, 1]
    for i in range(5):
        try:
            recip = 1.0 / values[i]
            print(f'reciprocal of {values[i]} at {i} is {recip}')
        except IndexError: # when i > 2
            print(f'index {i} out of range')
        except ZeroDivisionError:
            print('Division by zero!')
```

Example: `i == 0`

```
values = [-1, 0, 1]
for i in range(5): # (0, 1, 2, 3, 4)
    try:
        recip = 1.0 / values[i] # 1.0 / -1 = -1.0
        print(f'reciprocal of {values[i]} at {i} is {recip}')
    except IndexError: # when i > 2
        print(f'index {i} out of range')
    except ZeroDivisionError:
        print('Division by zero!')
```

This prints: `reciprocal of -1 at 0 is -1`

Example: `i == 1`

```
values = [-1, 0, 1]
for i in range(5): # (0, 1, 2, 3, 4)
    try:
        recip = 1.0 / values[i]  # 1.0 / 0
        print(f'reciprocal of {values[i]} at {i} is {recip}')
    except IndexError: # when i > 2
        print(f'index {i} out of range')
    except ZeroDivisionError:
        print('Division by zero!')
```

This prints: `Division by zero!`

Example: `i == 2`

```
values = [-1, 0, 1]
for i in range(5): # (0, 1, 2, 3, 4)
    try:
        recip = 1.0 / values[i]  # 1.0 / 1 = 1.0
        print(f'reciprocal of {values[i]} at {i} is {recip}')
    except IndexError: # when i > 2
        print(f'index {i} out of range')
    except ZeroDivisionError:
        print('Division by zero!')
```

This prints: `reciprocal of 1 at 2 is 1`

Example: `i == 3`

```
values = [-1, 0, 1]
for i in range(5): # (0, 1, 2, 3, 4)
    try:
        recip = 1.0 / values[i]  # values[3] does not exist
        print(f'reciprocal of {values[i]} at {i} is {recip}')
    except IndexError: # when i > 2
        print(f'index {i} out of range')
    except ZeroDivisionError:
        print('Division by zero!')
```

This prints: `index 3 out of range`

Raising Exceptions with `raise`

Use the **`raise`** statement if you want to raise your own exception:

```
>>> raise ValueError
```

This will result in:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError
```

Moving forward, we will be omitting `Traceback` components of error output.

Raising Exceptions with `raise`

You can add an error message to a `raise` statement with most built-in exceptions:

```
>>> raise ValueError('Invalid value')
```

This will result in:

```
ValueError: Invalid Value
```

This can be used to give more detail about what specific error happened

Example

Consider a function `net_cost(price, tax_amount)` which computes the net cost of an item as `price + (price * tax_amount)`. For example, `net_cost(1.00, 0.08)` would return `1.08` for an item with a price of $1.00 and 8% tax.

If either argument is `< 0`, this is invalid (but does not throw an exception when passed).

```
>>> net_cost(-50, 0.08)
-90.0 # !
>>> net_cost(-50, -0.08)
90.0  # !
```

Instead of just “hoping the client knows what to do”, we can raise our own `ValueError` exception if either argument is `< 0`

Example

```
def net_cost(price, tax):
    """
    Returns the net cost of an item with given `price`
    and `tax` rate.
    """
    if price < 0 or tax < 0:
        raise ValueError('price and tax must both be non-negative')
    return price + (price * tax)
```

Notice: the `raise` statement doesn't have to be in a `try`/`except` statement

Example

Using the `net_cost` function:

```
price = float(input('Enter a price: '))
tax_rate = float(input('Enter a tax rate: '))
try:
    cost = net_cost(price, tax_rate)
    print(f'The cost of your item is ${cost:.2f}.')
except ValueError as e: # e is the ValueError object that was raised
    print(e) # prints error message
```

Note: This is an example where the specific exception we're catching is not in the function the error is raised in, but in the code that *called* the function

New `except` Syntax

Here, we notice a new form of `except` block syntax:

```
try:
    # code that may raise an error
except ValueError as e:
    # code that uses the ValueError instance
```

The caught `ValueError` **object** is given a name (`e`) that can now be used within the `except` block as it sees fit (e.g. by printing)

`as` is another Python keyword - where have we seen it before?

```
import <module> as <name>
```

Slightly Different Example

We *could* also handle the exception directly in the `net_cost` function:

```
def net_cost(price, tax):
    """
    Returns the net cost of an item with given `price`
    and `tax` rate.
    """
    try:
        if price < 0 or tax < 0:
            raise ValueError('price and tax must both be non-negative.')
        return price + (price * tax)
    except ValueError as e:
        print(e)
```

Slightly Different Example

Handling an exception in the function that raised it is occasionally useful, but more often it is *not* the right thing to do

What we want to ask is:

- *Whose responsibility is it if this function fails?*

Slightly Different Example

For `net_cost`, whatever called net_cost was responsible for giving `net_cost` valid inputs

If that doesn't happen, it's not `net_cost`'s problem

- Just raise the exception and let other code handle it

We say that `net_cost` has a **precondition** that its inputs are both `>= 0`

If a precondition is violated, just inform the caller by raising an exception

Slightly Different Example

Code that called `net_cost` has many different options to handle the error:

- Print error message and quit
- Ask user for a different input
- Choose another value (default value)
- etc.

When the exception-raising function (e.g. `net_cost`) is not given sole responsibility for making these decisions, code can be much more flexible with more options for callers to choose from

Next, let's look more closely into how to create *our own* exceptions

Another Example: Error-Handling with Files

When we introduced files and opening them with **`open(filename, 'r')`**, we assumed that the files being read existed

When a file opened for reading does not exist, a **`FileNotFoundError`** exception is raised

This is very common, e.g. when a user inputs the wrong file name

How can we handle this in our code?

Let's write a function to read a file having a number per line, and summing all of the numbers.

First Attempt (Without Error-Handling)

```
def sum_numbers_in_file(filename):
    sum_nums = 0
    f = open(filename, 'r')
    for line in f:
        # str.strip() strips away any spaces and \n character
        sum_nums += int(line.strip())
    f.close()
    return sum_nums
```

This is ok, but it makes a lot of assumptions. What kind of errors could occur?

- The file might not exist
- The file may contain lines with non-numbers
- The file may contain lines with multiple numbers
- The file may have blank lines

Error-Handling

First, we need to know what kinds of Python **exceptions** are **raised** when these errors happen

If the file doesn't exist, a **`FileNotFoundError`** exception is raised when attempting to open the file

What about other errors?

Error-Handling

What if the line contains something other than numbers?

```
42          # OK
-1          # OK
forty-two   # Error: ValueError
            # Error: ValueError
1 2 3       # Error: ValeuError
10010       # OK
```

`ValueError` occurs when trying to execute this line::

```
sum_nums += int(line)
```

ValueError

```
sum_nums += int('forty-two')
```

Gives this error message:

```
ValueError: invalid literal for int() with base 10: 'forty-two'
```

Breaking it down:

- Exception type: **`ValueError`**
- The data associated with the exception gives more information about what caused the exception: **`invalid literal for int() with base 10: 'forty-two'`**

ValueError

What about blank lines?

```
sum_nums += int('')
```

results in:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
```

Python raises the same exception!

`ValueError`

What about lines with multiple numbers?

```
sum_nums += int('1 2 3')
```

results in:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1 2 3'
```

Python raises the same exception again! We can use this to group the mis-formatted lines with an exception case for **`ValueError`**

`FileNotFoundError` and `ValueError`

To summarize, the function **`sum_numbers_in_file`**:

- Takes in an input argument representing a file of numbers, one per line
- Outputs the sum of all numbers in the file
- May raise a **`FileNotFoundError`** if the file doesn't exist
- Otherwise, may raise a **`ValueError`** if the file isn't formatted correctly (non-numbers on blank lines or too many numbers on some lines)

We need to be able to handle two different kinds of error situations as well as the normal case

How many **`try`**/**`except`** blocks will we need?

Handling Exceptions

New version with exception handling:

```
def sum_of_numbers_in_file(filename):
    try:
        # Previous code
    except FileNotFoundError:
        # What to do here?
    except ValueError:
        # What to do here?
```

Handling a `FileNotFoundError`

Remember that there are multiple ways to handle any particular error

Error-handling in programs should always consider an intuitive and user-friendly way to handle incorrect input

What are some things we could do to handle a **`FileNotFoundError`** (i.e. when the argument is a filename that doesn't exist)?

- Interactively prompt the user for a different filename, or
- Immediately return 0 as the sum value since the filename is invalid
- Abort entirely (don't handle the exception) and let the program terminate

Handling a `ValueError`:

What are some things we could do to handle a **`ValueError`** (i.e. when the file is improperly formatted)?

- Assume the line is no good, just use 0 as the number and keep going
- Assume the entire file is corrupt, return 0 as the sum for the function
- Abort entirely (don't handle the exception) and let the program terminate

`sum_nums_in_file` (with Error-Handling)

```
def sum_nums_in_file(filename):
    while True:
        try:
            f = open(filename, 'r')
            # process sum for each line, return sum when done
        except FileNotFoundError:
            print(f'Couldn\'t read file: {filename}')
            filename = input('Enter another filename: ')
        except ValueError:
            print(f'Invalid line in file: {filename}')
            filename = input('Enter another filename: ')
```

Design Decisions...

There are many choices to make!

And one further question:

- Should this function be in charge of deciding what happens in the event of an error?

If reading in a small number of files, perhaps it's ok to prompt a user for a new filename in case the given one doesn't exist

If trying to read a large list of files, this would not be appropriate