Exceptions & Decorators
Class outline:

- Exceptions
- Decorators
Exceptions
Handling errors

Sometimes, computer programs behave in non-standard ways.

- A function receives an argument value of an improper type
- Some resource (such as a file) is not available
- A network connection is lost in the middle of data transmission

Moth found in a Mark II Computer (Grace Hopper's Notebook, 1947)
Exceptions

An **exception** is a built-in mechanism in a programming language to declare and respond to "exceptional" conditions.

A program raises an exception when an error occurs.

If the exception is not handled, the program will stop running entirely.

But if a programmer can anticipate when exceptions might happen, they can include code for **handling the exception**, so that the program continues running.

Many languages include exception handling: C++, Java, Python, JavaScript, etc.
Exceptions in Python

Python raises an exception whenever a runtime error occurs.

How an unhandled exception is reported:

```python
>>> 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in
ZeroDivisionError: division by zero
```

If an exception is not handled, the program stops executing immediately.
### Types of exceptions

A few exception types and examples of buggy code:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>OverflowError</td>
<td><code>pow(2.12, 1000)</code></td>
</tr>
<tr>
<td>TypeError</td>
<td><code>'hello'[1] = 'j'</code></td>
</tr>
<tr>
<td>IndexError</td>
<td><code>'hello'[7]</code></td>
</tr>
<tr>
<td>NameError</td>
<td><code>x += 5</code></td>
</tr>
<tr>
<td>FileNotFoundError</td>
<td><code>open('dsfdfd.txt')</code></td>
</tr>
</tbody>
</table>

See full list in the exceptions docs.
The try statement

To handle an exception (keep the program running), use a try statement.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

The <try suite> is executed first. If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and if the class of the exception inherits from <exception class>, then the <except suite> is executed, with <name> bound to the exception.
Try statement example

```python
try:
    quot = 10/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    quot = 0
```

Try in PythonTutor
Try inside a function

def div_numbers(dividend, divisor):
    try:
        quotient = dividend/divisor
    except ZeroDivisionError:
        print("Function was called with 0 as divisor")
        quotient = 0
    return quotient

div_numbers(10, 2)
div_numbers(10, 0)
div_numbers(10, -1)

Try in PythonTutor
def invert(x):
    inverse = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return inverse

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        print('Handled', e)
        return 0
What would Python Do?

```python
def invert(x):
    inverse = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return inverse

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        print('Handled', e)
        return 0

invert_safe(1/0)
```
def invert(x):
    inverse = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return inverse

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        print('Handled', e)
        return 0

invert_safe(1/0)

try:
    invert_safe(0)
except ZeroDivisionError as e:
    print('Handled!')
def invert(x):
    inverse = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return inverse

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        print('Handled', e)
    return 0

invert_safe(1/0)

try:
    invert_safe(0)
except ZeroDivisionError as e:
    print('Handled!')

inverrrrt_safe(1/0)
Raising exceptions
Assert statements

Assert statements raise an exception of type `AssertionError`:

```
assert <expression>, <string>
```

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag; "O" stands for optimized.

```
python3 -O
```
Raise statements

Any type of exception can be raised with a `raise` statement

```python
raise <expression>
```

`<expression>` must evaluate to a subclass of `BaseException` or an instance of one

Exceptions are constructed like any other object. E.g.,

```python
TypeError('Bad argument!')
```
Decorators
def trace1(f):
    """Return a function that takes a single argument, x, prints it,
    computes and prints F(x), and returns the computed value.
    >>> square = lambda x: x * x
    >>> trace1(square)(3)
    -> 3
    <- 9
    9
    """
A tracing function

Let's make a higher-order tracing function.

```python
def trace1(f):
    """Return a function that takes a single argument, x, prints it, computes and prints F(x), and returns the computed value.
    >>> square = lambda x: x * x
    >>> trace1(square)(3)
    -> 3
    <- 9
    9
    """
    def traced(x):
        print("->", x)
        r = f(x)
        print("<-", r)
        return r
    return traced
```
A tracing decorator

What if we always wanted a function to be traced?

```python
@trace1
def square(x):
    return x * x
```

That's equivalent to..

```python
def square(x):
    return x * x
square = trace1(square)
```
General decorator syntax

The notation:

```
@ATTR
def aFunc(...):
    ...
```

is essentially equivalent to:

```
def aFunc(...):
    ...
    aFunc = ATTR(aFunc)
```

**ATTR** can be any expression, not just a single function name.