MODULE-5

OBJECT ORIENTED PROGRAMMING, INHERITANCE AND EXCEPTIONS

SYLLABUS:
Object oriented programming: *Objects are mutable, Sameness, Copying.*
Inheritance: *Pure functions, Modifiers, Generalization, Operator Overloading, Polymorphism.*
Exceptions: *Catching Exceptions, Raising your own exceptions.*

CHAPTERS: *11.2.2-11.2.4, 11.3.2-11.3.9, 12.1, 12.2*

TEXT BOOK: *Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers- How to think like a computer scientist: learning with python 3. Green Tea Press, Wellesley, Massachusetts, 2020*

Lecture - 33

OBJECTS ARE MUTABLE

- We can change the state of an object by making an assignment to one of its attributes. For example, to grow the size of a rectangle without changing its position, we could modify the values of width and height:

```
box.width += 50
box.height += 100
```

- Of course, we'd probably like to provide a method to encapsulate this inside the class. We will also provide another method to move the position of the rectangle elsewhere:

```
class Rectangle:
    # ...
    def grow(self, delta_width, delta_height):
        """ Grow (or shrink) this object by the deltas """
        self.width += delta_width
        self.height += delta_height
    def move(self, dx, dy):
        """ Move this object by the deltas """
        self.corner.x += dx
        self.corner.y += dy
```

- Let us try this:

```
>>> r = Rectangle(Point(10,5), 100, 50)
>>> print(r)
((10, 5), 100, 50)
>>> r.grow(25, -10)
>>> print(r)
((10, 5), 125, 40)
>>> r.move(-10, 10)
print(r)
```

```
((0, 15), 125, 40)
```

SAMENESS

- The meaning of the word "same" seems perfectly clear until we give it some thought, and then we realize there is more to it than we initially expected.
- For example, if we say, "Alice and Bob have the same car", we mean that her car and his are the same make and model, but that they are two different cars. If we say, "Alice and Bob have the same mother", we mean that her mother and his are the same person.
- When we talk about objects, there is a similar ambiguity. For example, if two Points are the same, does that mean they contain the same data (coordinates) or that they are actually the same object?
- We've already seen the is operator in the chapter on lists, where we talked about aliases: it allows us to find out if two references refer to the same object:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> p1 is p2
False
```

- Even though p1 and p2 contain the same coordinates, they are not the same object. If we assign p1 to p3, then the two variables are aliases of the same object:

```
>>> p3 = p1
>>> p1 is p3
True
```

- This type of equality is called shallow equality because it compares only the references, not the contents of the objects. To compare the contents of the objects deep equality, we can write a function called same_coordinates:

```
def same_coordinates(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)
```

- Now if we create two different objects that contain the same data, we can use same_point to find out if they represent points with the same coordinates.

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> same_coordinates(p1, p2)
True
```

- Of course, if the two variables refer to the same object, they have both shallow and deep equality.

Beware of ==

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean — neither more nor less." Alice in Wonderland

Python has a powerful feature that allows a designer of a class to decide what an operation like == or < should mean. (We've just shown how we can control how our own objects are converted to strings, so we've

already made a start!) We'll cover more detail later. But sometimes the implementors will attach shallow equality semantics, and sometimes deep equality, as shown in this little experiment:

```
p = Point(4, 2)
s = Point(4, 2)
print("== on Points returns", p == s)
# By default, == on Point objects does a shallow equality test

a = [2,3]
b = [2,3]
print("== on lists returns", a == b)
# But by default, == does a deep equality test on lists
```

This outputs:

```
== on Points returns False
== on lists returns True
```

So, we conclude that even though the two lists (or tuples, etc.) are distinct objects with different memory addresses, for lists the == operator tests for deep equality, while in the case of points it makes a shallow test.

Review questions:

1. What does it mean for an object to be mutable?
2. What is sameness?
3. What is the default behavior of the == operator for lists?
4. What is the difference between shallow equality and deep equality?

Lecture - 34

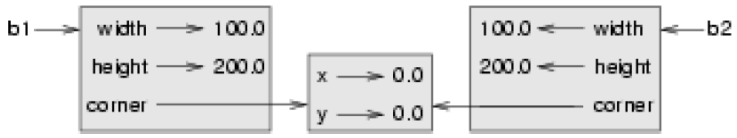
COPYING

- Aliasing can make a program difficult to read because changes made in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.
- Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
>>> import copy
>>> p1 = Point(3, 4)
>>> p2 = copy.copy(p1)
>>> p1 is p2
False
>>> same_coordinates(p1, p2)
True
```

- Once we import the copy module, we can use the copy function to make a new Point. p1 and p2 are not the same point, but they contain the same data.
- To copy a simple object like a Point, which doesn't contain any embedded objects, copy is sufficient. This is called shallow copying.

- For something like a Rectangle, which contains a reference to a Point, copy doesn't do quite the right thing. It copies the reference to the Point object, so both the old Rectangle and the new one refer to a single Point.
- If we create a box, b1, in the usual way and then make a copy, b2, using copy, the resulting state diagram looks like this:



- This is almost certainly not what we want. In this case, invoking grow on one of the Rectangle objects would not affect the other, but invoking move on either would affect both! This behavior is confusing and error-prone. The shallow copy has created an alias to the Point that represents the corner.
- Fortunately, the copy module contains a function named deepcopy that copies not only the object but also any embedded objects. It won't be surprising to learn that this operation is called a deep copy.

```
>>> b2 = copy.deepcopy(b1)
```

- Now b1 and b2 are completely separate objects.

Review questions:

1. What is the purpose of the copy module in Python?
2. What is shallow copying?
3. What is deep copying?
4. What is the difference between shallow copy and deep copy?

Lecture - 35

PURE FUNCTIONS

- Here, we'll write two versions of a function called add_time, which calculates the sum of two MyTime objects. They will demonstrate two kinds of functions: pure functions and modifiers.
- The following is a rough version of add_time:

```
def add_time(t1, t2):
    h = t1.hours + t2.hours
    m = t1.minutes + t2.minutes
    s = t1.seconds + t2.seconds
    sum_t = MyTime(h, m, s)
    return sum_t
```

- The function creates a new MyTime object and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as parameters and it has no side effects, such as updating global variables, displaying a value, or getting user input.
- Here is an example of how to use this function. We'll create two MyTime objects: current_time, which contains the current time; and bread_time, which contains the amount of time it takes for a breadmaker to make bread. Then we'll use add_time to figure out when the bread will be done.

```
>>> current_time = MyTime(9, 14, 30)
>>> bread_time = MyTime(3, 35, 0)
>>> done_time = add_time(current_time, bread_time)
>>> print(done_time)
12:49:30
```

- The output of this program is 12:49:30, which is correct. On the other hand, there are cases where the result is not correct.
- The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to carry the extra seconds into the minutes column or the extra minutes into the hours column.
- Here's a better version of the function:

```
def add_time(t1, t2):
    h = t1.hours + t2.hours
    m = t1.minutes + t2.minutes
    s = t1.seconds + t2.seconds
    if s >= 60:
        s -= 60
        m += 1
    if m >= 60:
        m -= 60
        h += 1
    sum_t = MyTime(h, m, s)
    return sum_t
```

- This function is starting to get bigger, and still doesn't work for all possible cases.

Review questions:

1. What is a pure function?
2. What does the add_time function return?
3. What kind of side effects does a pure function avoid?
4. What is the purpose of creating a new MyTime object inside the add_time function?

Lecture - 36

MODIFIERS

- There are times when it is useful for a function to modify one or more of the objects it gets as parameters. Usually, the caller keeps a reference to the objects it passes, so any changes the function makes are visible to the caller. Functions that work this way are called modifiers.
- increment, which adds a given number of seconds to a MyTime object, would be written most naturally as a modifier. A rough draft of the function looks like this:

```
def increment(t, secs):
    t.seconds += secs

    if t.seconds >= 60:
        t.seconds -= 60
        t.minutes += 1
```

```
    if t.minutes >= 60:
      t.minutes -= 60
      t.hours += 1
```

- The first line performs the basic operation; the remainder deals with the special cases. Is this function correct? What happens if the parameter seconds is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until seconds is less than sixty. One solution is to replace the if statements with while statements:

```
def increment(t, seconds):
    t.seconds += seconds

    while t.seconds >= 60:
            t.seconds -= 60
            t.minutes += 1

    while t.minutes >= 60:
            t.minutes -= 60
            t.hours += 1
```

- This function is now correct when seconds is not negative, and when hours does not exceed 23, but it is not a particularly good solution.

CONVERTING INCREMENT TO A METHOD

- OOP programmers would prefer to put functions that work with MyTime objects into the MyTime class, so let's convert increment to a method. To save space, we will leave out previously defined methods, but you should keep them in your version:

```
class MyTime:
    # Previous method definitions here...

    def increment(self, seconds):
        self.seconds += seconds

        while self.seconds >= 60:
            self.seconds -= 60
            self.minutes += 1

        while self.minutes >= 60:
            self.minutes -= 60
            self.hours += 1
```

- The transformation is purely mechanical — we move the definition into the class definition and (optionally) change the name of the first parameter to self, to fit with Python style conventions. Now we can invoke increment using the syntax for invoking a method.

```
current_time.increment(500)
```

- Again, the object on which the method is invoked gets assigned to the first parameter, self. The second parameter, seconds gets the value 500.

An "Aha!" insight

Often a high-level insight into the problem can make the programming much easier. In this case, the insight is that a MyTime object is really a three-digit number in base 60! The second component is the ones column, the minute component is the sixties column, and the hour component is the thirty-six hundreds column.

When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem — we can convert a MyTime object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The following method is added to the MyTime class to convert any instance into a corresponding number of seconds:

```
class MyTime:
    # ...

    def to_seconds(self):
        """ Return the number of seconds represented
            by this instance
        """
        return self.hours * 3600 + self.minutes * 60 + self.seconds
```

Now, all we need is a way to convert from an integer back to a MyTime object. Supposing we have tsecs seconds, some integer division and mod operators can do this for us:

```
hrs = tsecs // 3600
leftoversecs = tsecs % 3600
mins = leftoversecs // 60
secs = leftoversecs % 60
```

You might have to think a bit to convince yourself that this technique to convert from one base to another is correct.

In OOP we're really trying to wrap together the data and the operations that apply to it. So we'd like to have this logic inside the MyTime class. A good solution is to rewrite the class initializer so that it can cope with initial values of seconds or minutes that are outside the normalized values. (A normalized time would be something like 3 hours 12 minutes and 20 seconds. The same time, but unnormalized could be 2 hours 70 minutes and 140 seconds.)

Let's rewrite a more powerful initializer for MyTime:

```
class MyTime:
# ...

def __init__(self, hrs=0, mins=0, secs=0):
    """ Create a new MyTime object initialized to hrs, mins, secs.
        The values of mins and secs may be outside the range 0-59,
        but the resulting MyTime object will be normalized.
    """

      # Calculate total seconds to represent
      totalsecs = hrs*3600 + mins*60 + secs
```

```
    self.hours = totalsecs // 3600 # Split in h, m, s
    leftoversecs = totalsecs % 3600
    self.minutes = leftoversecs // 60
    self.seconds = leftoversecs % 60
```

Now we can rewrite add_time like this:

```
def add_time(t1, t2):
    secs = t1.to_seconds() + t2.to_seconds()
    return MyTime(0, 0, secs)
```

This version is much shorter than the original, and it is much easier to demonstrate or reason that it is correct.

Review questions:

1. What is a modifier function?
2. What does the increment function do to a MyTime object?
3. What is the purpose of the self parameter in a method?
4. How does a modifier affect the object passed to it?

Lecture - 37

GENERALIZATION

- In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.
- But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversions, we get a program that is shorter, easier to read and debug, and more reliable.
- It is also easier to add features later. For example, imagine subtracting two MyTime objects to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.
- Ironically, sometimes making a problem harder (or more general) makes the programming easier, because there are fewer special cases and fewer opportunities for error.

Specialization versus Generalization

Computer Scientists are generally fond of specializing their types, while mathematicians often take the opposite approach, and generalize everything.

What do we mean by this?

If we ask a mathematician to solve a problem involving weekdays, days of the century, playing cards, time, or dominoes, their most likely response is to observe that all these objects can be represented by integers. Playing cards, for example, can be numbered from 0 to 51. Days within the century can be numbered. Mathematicians will say "These things are enumerable—the elements can be uniquely numbered (and we can reverse this numbering to get back to the original concept). So let's number them, and confine our thinking to integers. Luckily, we have powerful techniques and a good understanding of integers, and so our abstractions — the way we tackle and simplify these problems — is to try to reduce them to problems about integers."

Computer Scientists tend to do the opposite. We will argue that there are many integer operations that are simply not meaningful for dominoes, or for days of the century. So we'll often define new specialized types, like MyTime, because we can restrict, control, and specialize the operations that are possible. Object-oriented programming is particularly popular because it gives us a good way to bundle methods and specialized data into a new type.

Both approaches are powerful problem-solving techniques. Often it may help to try to think about the problem from both points of view—"What would happen if I tried to reduce everything to very few primitive types?", versus "What would happen if this thing had its own specialized type?"

ANOTHER EXAMPLE

- The after function should compare two times, and tell us whether the first time is strictly after the second, e.g.

```
>>> t1 = MyTime(10, 55, 12)
>>> t2 = MyTime(10, 48, 22)
>>> after(t1, t2) # Is t1 after t2?
True
```

- This is slightly more complicated because it operates on two MyTime objects, not just one. But we'd prefer to write it as a method anyway— in this case, a method on the first argument:

```
class MyTime:
    # Previous method definitions here...

    def after(self, time2):
        """ Return True if I am strictly greater than time2 """
        if self.hours > time2.hours:
            return True
        if self.hours < time2.hours:
            return False
        if self.minutes > time2.minutes:
            return True
        if self.minutes < time2.minutes:
            return False
        if self.seconds > time2.seconds:
            return True

        return False
```

- We invoke this method on one object and pass the other as an argument:

```
if current_time.after(done_time):
    print("The bread will be done before it starts!")
```

- The logic of the if statements deserve special attention here. Lines 11-18 will only be reached if the two hour fields are the same. Similarly, the test at line 16 is only executed if both times have the same hours and the same minutes.

- Could we make this easier by using our "Aha!" insight and extra work from earlier, and reducing both times to integers? Yes, with spectacular results!

```
class MyTime:
    # Previous method definitions here...

    def after(self, time2):
        """ Return True if I am strictly greater than time2 """
        return self.to_seconds() > time2.to_seconds()
```

- This is a great way to code this: if we want to tell if the first time is after the second time, turn them both into integers and compare the integers.

Review questions:

1. What is generalization?
2. How does generalizing a problem reduce special cases and errors?
3. What is the main advantage of making a problem more general before programming?
4. How does specialization differ from generalization?

Lecture - 38

OPERATOR OVERLOADING

- Some languages, including Python, make it possible to have different meanings for the same operator when applied to different types. For example, + in Python means quite different things for integers and for strings. This feature is called operator overloading.
- It is especially useful when programmers can also overload the operators for their own user-defined types. For example, to override the addition operator +, we can provide a method named __add__:

```
class MyTime:
    # Previously defined methods here...
    def __add__(self, other):
        return MyTime(0, 0, self.to_seconds() + other.to_seconds())
```

- As usual, the first parameter is the object on which the method is invoked. The second parameter is conveniently named other to distinguish it from self. To add two MyTime objects, we create and return a new MyTime object that contains their sum.
- Now, when we apply the + operator to MyTime objects, Python invokes the __add__ method that we have written:

```
>>> t1 = MyTime(1, 15, 42)
>>> t2 = MyTime(3, 50, 30)
>>> t3 = t1 + t2
>>> print(t3)
05:06:12
```

- The expression t1 + t2 is equivalent to t1.__add__(t2), but obviously more elegant. As an exercise, add a method __sub__(self, other) that overloads the subtraction operator, and try it out.

- For the next couple of exercises we'll go back to the Point class defined in our first chapter about objects, and overload some of its operators. Firstly, adding two points adds their respective (x, y) coordinates:

```
class Point:
    # Previously defined methods here...
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

- There are several ways to override the behavior of the multiplication operator: by defining a method named __mul__, or __rmul__, or both. It computes the dot product of the two Points, defined according to the rules of linear algebra:

```
def __mul__(self, other):
    return self.x * other.x + self.y * other.y
```

- If the left operand of * is a primitive type and the right operand is a Point, Python invokes __rmul__, which performs scalar multiplication:

```
def __rmul__(self, other):
    return Point(other * self.x, other * self.y)
```

- The result is a new Point whose coordinates are a multiple of the original coordinates. If other is a type that cannot be multiplied by a floating-point number, then __rmul__ will yield an error. This example demonstrates both kinds of multiplication:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print(p1 * p2)
43
>>> print(2 * p2)
(10, 14)
```

- What happens if we try to evaluate p2 * 2? Since the first parameter is a Point, Python invokes __mul__ with 2 as the second argument. Inside __mul__, the program tries to access the x coordinate of other, which fails because an integer has no attributes:

```
>>> print(p2 * 2)
AttributeError: 'int' object has no attribute 'x'
```

- Unfortunately, the error message is a bit opaque. This example demonstrates some of the difficulties of object-oriented programming. Sometimes it is hard enough just to figure out what code is running.

Review questions:

1. What is operator overloading?
2. How do you overload the addition operator + for a user-defined class like MyTime?
3. What is the advantage of overloading operators for user-defined types?
4. What is the difference between __mul__ and __rmul__?

Lecture - 39

POLYMORPHISM

- Most of the methods we have written only work for a specific type. When we create a new object, we write methods that operate on that type.
- But there are certain operations that we will want to apply to many types, such as the arithmetic operations in the previous sections. If many types support the same set of operations, we can write functions that work on any of those types.
- For example, the multadd operation (which is common in linear algebra) takes three parameters; it multiplies the first two and then adds the third. We can write it in Python like this:

```
def multadd (x, y, z):
    return x * y + z
```

- This function will work for any values of x and y that can be multiplied and for any value of z that can be added to the product. We can invoke it with numeric values:

```
>>> multadd (3, 2, 1)
7
```

or with Points:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print(multadd (2, p1, p2))
(11, 15)
>>> print(multadd (p1, p2, 1))
44
```

- In the first case, the Point is multiplied by a scalar and then added to another Point. In the second case, the dot product yields a numeric value, so the third parameter also has to be a numeric value. A function like this that can take arguments with different types is called polymorphic.
- As another example, consider the function front_and_back, which prints a list twice, forward and backward:

```
def front_and_back(front):
    import copy
    back = copy.copy(front)
    back.reverse()
    print(str(front) + str(back))
```

- Because the reverse method is a modifier, we make a copy of the list before reversing it. That way, this function doesn't modify the list it gets as a parameter. Here's an example that applies front_and_back to a list:

```
>>> my_list = [1, 2, 3, 4]
>>> front_and_back(my_list)
[1, 2, 3, 4][4, 3, 2, 1]
```

- Of course, we intended to apply this function to lists, so it is not surprising that it works. What would be surprising is if we could apply it to a Point.
- To determine whether a function can be applied to a new type, we apply Python's fundamental rule of polymorphism, called the duck typing rule: If all of the operations inside the function can be applied to the type, the function can be applied to the type. The operations in the front_and_back function include copy, reverse, and print.
- Not all programming languages define polymorphism in this way. Look up duck typing, and see if you can figure out why it has this name.
- Copy works on any object, and we have already written a __str__ method for Point objects, so all we need is a reverse method in the Point class:

```
def reverse(self):
    (self.x , self.y) = (self.y, self.x)
```

- Then we can pass Points to front_and_back:

```
>>> p = Point(3, 4)
>>> front_and_back(p)
(3, 4)(4, 3)
```

- The most interesting polymorphism is the unintentional kind, where we discover that a function, we have already written can be applied to a type for which we never planned.

Review questions:

1. What is polymorphism?
2. What does it mean for a function to be polymorphic?
3. What is the duck typing rule in Python?
4. Why is polymorphism useful when writing functions for multiple types?

Lecture - 40

CATCHING EXCEPTIONS

- Whenever a runtime error occurs, it creates an exception object. The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred.
- For example, dividing by zero creates an exception:

```
>>> print(55/0)
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- So does accessing a non-existent list item:

```
>>> a = []
>>> print(a[5])
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

- Or trying to make an item assignment on a tuple:

```
>>> tup = ("a", "b", "d", "d")
>>> tup[2] = "c"
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- In each case, the error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon.
- Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop. We can handle the exception using the try statement to "wrap" a region of code.
- For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
filename = input("Enter a file name: ")
try:
    f = open(filename, "r")
except FileNotFoundError:
     print("There is no file named", filename)
```

- The try statement has four separate clauses—or parts—introduced by the keywords try, except, else, and finally. All clauses but the try can be omitted.
- The interpretor executes the block under the try statement, and monitors for exceptions. If one occurs, the interpretor moves to the except statement; it executes the expect block if the exception raised match the exception requested in the except statement.
- If no exception occurs, the interpretor skips the block under the except clause. A else block is executed after the try one, if no exception occurred. A finally block is executed in any case. With all the statements, a try clause looks like:

```
user_input = input('Type a number:')
try:
    # Try do do something that could fail.
    user_input_as_number = float(user_input)
except ValueError:
     # This will be executed if a ``ValueError`` is raised.
     print('You did not enter a number.')
else:
    # This will be executed if not exception got raised in the
    # ``try`` statement.
    print('The square of your number is ', user_input_as_number**2)
finally:
    # This will be executed whether or not an exception is raised.
    print('Thank you')
```

- When using a try clause, you should have as little as possible in the try block. If too many things happen in that block, you risk handling an unexpected exception. If the try block can fail if various way, you can handle different exceptions in the same try clause:

```
try:
    with open(filename) as infile:
        content = infile.read()
```

```
except FileNotFoundError:
    print('The file does not exist.')
except PermissionError:
    print('Your are not allowed to read this file.')
```

- It is also possible not to specify a particular exception in the except statement. In this case, any exception will be handled. Such bare except statement should be avoided, though, as they can easily mask bugs.

RAISING OUR OWN EXCEPTIONS

- Can our program deliberately cause its own exceptions? If our program detects an error condition, we can raise an exception. Here is an example that gets input from the user and checks that the number is non-negative:

```
def get_age():
    age = int(input("Please enter your age: "))
    if age < 0:
        # Create a new instance of an exception
        my_error = ValueError("{0} is not a valid age".format(age))
        raise my_error
    return age
```

- Line 5 creates an exception object, in this case, a ValueError object, which encapsulates specific information about the error. Assume that in this case function A called B which called C which called D which called get_age. The raise statement on line 6 carries this object out as a kind of "return value", and immediately exits from get_age() to its caller D.
- Then D again exits to its caller C, and C exits to B and so on, each returning the exception object to their caller, until it encounters a try ... except that can handle the exception. We call this "unwinding the call stack".
- ValueError is one of the built-in exception types which most closely matches the kind of error we want to raise. The complete listing of built-in exceptions can be found at the Built-in Exceptions section of the Python Library Reference , again by Python's creator, Guido van Rossum.
- If the function that called get_age (or its caller, or their caller, . . .) handles the error, then the program can carry on running; otherwise, Python prints the traceback and exits:

```
>>> get_age()
Please enter your age: 42
42
>>> get_age()
Please enter your age: -2
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "learn_exceptions.py", line 4, in get_age
    raise ValueError("{0} is not a valid age".format(age))
ValueError: -2 is not a valid age
```

- The error message includes the exception type and the additional information that was provided when the exception object was first created.
- It is often the case that lines 5 and 6 (creating the exception object, then raising the exception) are combined into a single statement, but there are really two different and independent things happening,

so perhaps it makes sense to keep the two steps separate when we first learn to work with exceptions. Here we show it all in a single statement:

```
raise ValueError("{0} is not a valid age".format(age))
```

Review questions:

1. What is exception?
2. What is the purpose of a try block in Python?
3. Name the four clauses that can be used in a try statement.
4. What does the finally clause do in a try statement?

Question Bank

1. How objects are mutable? Example with an example.
2. Explain the following with an example:
   a. Sameness
   b. Copying
   c. Pure Functions
   d. Modifiers
3. What is operator overloading? Give an example.
4. Demonstrate Polymorphism with an example.
5. What do you mean by catching exceptions?
6. How to raise your own exceptions?
7. Add a method area to the Rectangle class that returns the area of any instance:
   r = Rectangle(Point(0, 0), 10, 5)
   test(r.area() == 50)
8. Create some test cases for the increment method. Consider specifically the case where the number of seconds to add to the time is negative. Fix up increment so that it handles this case if it does not do so already. (You may assume that you will never subtract more seconds than are in the time object.)
9. Write a function named readposint that uses the input dialog to prompt the user for a positive integer and then checks the input to confirm that it meets the requirements. It should be able to handle inputs that cannot be converted to int, as well as negative ints, and edge cases (e.g. when the user closes the dialog, or does not enter anything at all.)