In his Pulitzer-prizewinning book, computer scientist Douglas Hofstadter identifies two concepts—holism and reductionism—that turn out to be important as you begin to learn about programming. Hofstadter explains these concepts using a dialogue in the style of Lewis Carroll:

Achilles: I will be glad to indulge both of you, if you will first oblige me, by telling me the meaning of these strange expressions, “holism” and “reductionism”.

Holism is the most natural thing in the world to grasp. It’s simply the belief that “the whole is greater than the sum of its parts”. No one in his right mind could reject holism.

Anteater: Reductionism is the most natural thing in the world to grasp. It’s simply the belief that “a whole can be understood completely if you understand its parts, and the nature of their ‘sum’”. No one in her left brain could reject reductionism.

A Quick Review of Functions

- You have been working with functions ever since Chapter 1.
- At the most basic level, a function is a sequence of statements that has been collected together and given a name. The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the function name.
- The following terms are useful when working with functions:
  - Invoking a function by name is known as calling that function.
  - The caller passes information to a function using arguments.
  - When a function completes its operation, it returns to its caller.
  - A function gives information to the caller by returning a result.

Review: Syntax of Functions

- The general form of a function definition in Python is

```python
def name(parameter list):
    statements in the function body
```

where `name` is the name of the function, and `parameter list` is a list of variables used to hold the values of each argument.

- You can return a value from a function by including one or more `return` statements, which are usually written as

```python
return expression
```

where `expression` is an expression that specifies the value you want to return.

Predicate Functions

- Functions in Python can return values of any type. Those that return Boolean values play a central role in programming and are called predicate functions. As an example, the following function tests if the first argument is divisible by the second:

```python
def isDivisibleBy(x, y):
    return x % y == 0
```

- Once you have defined a predicate function, you can use it in any conditional expression. For example, you can print the integers between 1 and 99 that are divisible by 7 as follows:

```python
for i in range(1, 100):
    if isDivisibleBy(i, 7):
        print(i)
```

Using Predicate Functions Effectively

- New programmers often seem uncomfortable with Boolean values and end up writing really ugly code. For example, a beginner might write `isDivisibleBy` like this:

```python
def isDivisibleBy(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

While this code is not technically incorrect, it is sufficiently inelegant to deserve the bug symbol.

- A similar problem occurs when novices explicitly check to see whether a predicate function returns True. You should be careful to avoid such redundant tests in your own programs.
Exercise: Finding Perfect Numbers

- Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of \( n \) is any divisor less than \( n \) itself). They called such numbers perfect numbers. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.
- For the next several minutes of class, we're going to design and implement a Python program that finds all the perfect numbers between two limits. For example, if the limits are 1 and 10000, the output should look like this:
Extending `createFilledCircle`

- The next two lines walk through the steps involved in calling an extended version of the `createFilledCircle` function from Chapter 3. The new version has the following features:
  - Callers can leave out the fill color, in which case it defaults to "Black" (as shown on an earlier slide).
  - The fill and border colors may be specified independently.
  - The caller can use keyword arguments to specify the border and fill colors in either order.
- The code also illustrates the use of the `is` operator to test whether the value of the border parameter is `None`.
- Your task in these slides is to anticipate what happens next at each point in the function calling process.

---

Passing Parameters with Defaults

```python
xcenter = gw.getWidth() / 2
ycenter = gw.getHeight() / 2
radius = 3 / 8 * gw.getHeight()

# Add a circle with default fill color
gw.add(createFilledCircle(xcenter, ycenter, radius, "Red"))
```

```python
# Create a function with default arguments
def createFilledCircle(x, y, r, fill="Black", border=None):
    circle = Oval(x - r, y - r, 2 * r, 2 * r)
    if border is None:
        circle.setColor(fill)
        circle.setFillColor(fill)
    else:
        circle.setColor(border)
        circle.setFillColor(fill)
    return circle
```

Passing Keyword Parameters

```python
gw.add(createFilledCircle(250, 100, 75, border="Red", fill="Blue"))
```

```python
def createFilledCircle(x, y, r, fill="Black", border=None):
    if border is None:
        circle = Oval(x - r, y - r, 2 * r, 2 * r)
        circle.setColor(fill)
        circle.setFillColor(fill)
    else:
        circle.setColor(border)
        circle.setFillColor(fill)
    return circle
```

The Combinations Function

- To illustrate function calls, the text uses a function $C(n,k)$ that computes the combinations function, which is the number of ways one can select $k$ elements from a set of $n$ objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar.

How many ways are there to select two coins?

- penny + nickel
- penny + dime
- penny + quarter
- penny + dollar
- nickel + dime
- nickel + quarter
- nickel + dollar
- dime + quarter
- dime + dollar
- quarter + dollar

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula $C(n,k) = \frac{n!}{k! \times (n-k)!}$.
- Given that you already have a `fact` function, is easy to turn this formula directly into a function, as follows:

```python
def combinations(n, k):
    return fact(n) // (fact(k) * fact(n - k))
```

```python
# The next slide simulates the operation of combinations and fact in the context of a call to combinations(6, 2).
```

Tracing the Combinations Function

```python
from combinations import combinations

# Call combinations function
combinations(6, 2)
```

```python
# Python code to simulate the operation of combinations and fact
```

```python
def combinations(n, k):
    def fact(n):
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

    result = 1
    for i in range(1, n + 1):  
        result *= i
    return result

# Call the function with n=6 and k=2
combinations(6, 2)
```
Libraries and Interfaces

- Modern programming depends on the use of libraries. When you create a program, you write only a fraction of the code.
- Libraries can be viewed from two perspectives. Code that uses a library is called a **client**. The code for the library itself is called the **implementation**.
- The point at which the client and the implementation meet is called the **interface**, which serves as both a barrier and a communication channel:

```

client  implementation
```

The **gtools** Library

```
# File: gtools.py

from gtools import GRect, Goval, GLabel

def createFilledRect(x, y, width, height, fill='black', border='none'):
    """
    Creates a GRect filled with the specified fill color.
    If border is specified, the border appears in that color.
    """
    rect = GRect(x, y, width, height)
    rect.setFilled(True, fill)
    rect.setBorder(border)
    return rect

def createFilledCircle(x, y, r, fill='black', border='none'):
    """
    Creates a circle of radius r centered at the point (x, y) with the specified fill color.
    If border is specified, the border appears in that color.
    """
    circle = Goval(x - r, y - r, 2 * r, 2 * r)
    circle.setFilled(True)
    if border is None:
        circle.setColor(fill)
    else:
        circle.setColor(border)
        circle.setFillColor(fill)
    return circle

def createCenteredLabel(text, x, y, font='none'):
    """
    Creates a new GLabel centered at point (x, y) in both the horizontal and vertical directions.
    If font is specified, it is used to set the font of the label.
    """
    label = GLabel(text)
    if font is not None:
        label.setFont(font)
        label.setLocation(x - label.getWidth() / 2, y + label.getHeight() / 2)
    return label
```