

Chapter Five: Functions

REVISED SPRING 2023 – A.M.TSAASAN W/ G. ERNSBERGER

Chapter Goals

- To be able to implement functions
- To become familiar with the concept of parameter passing
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To learn how to think recursively
- Use [David Kay's Design Recipe](#) to build good functions

In this chapter, you will learn how to design and implement your own functions

Using the process of stepwise refinement, you will be able to break up complex tasks into sets of cooperating functions

But first... review...

And a little about

`break()`

`continue()`

`else()` on loops... weird, and only for fun

<https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops>

What about `quit()` and `exit()`?

These ends the module you are in... it dies a targeted death, not just your loop.

And `sys.exit()`?

This ends the whole program... it dies a powerful death all of it dead.

What is the output?

```
str1 = "helicopter"  
str2 = "pilot"  
str3 = str1 + str2  
print(str3)
```

- A. helicopter
- B. pilot
- C. helicopterpilot
- D. helicopter pilot
- E. Something else

What is the output?

```
str1 = "helicopter"  
str2 = " pilot"  
str3 = str1 + str2  
print(str3[12])
```

- A. p
- B. i
- C. l
- D. Array index out of bounds
- E. Something else

What is the output?

```
str1 = "helicopter"  
str2 = " pilot"  
str3 = str1[4]+str2[3:]  
print(str3[2])
```

- A. t
- B. ter
- C. terilo
- D. terlot
- E. Something else

What is the output?

```
input_string = "pilot"  
for i in range(len(input_string)):  
    print(input_string[i], end = "")
```

- A. pilot
- B. piilloot
- C. piilloott
- D. ppiilloott
- E. Something else

What is the output?

```
str1 = "pilot"  
str2 = "PILOT"  
print(str1 == str2)
```

- A. pilot
- B. PILOT
- C. True
- D. False
- E. Something looks wrong

Functions as Black Boxes

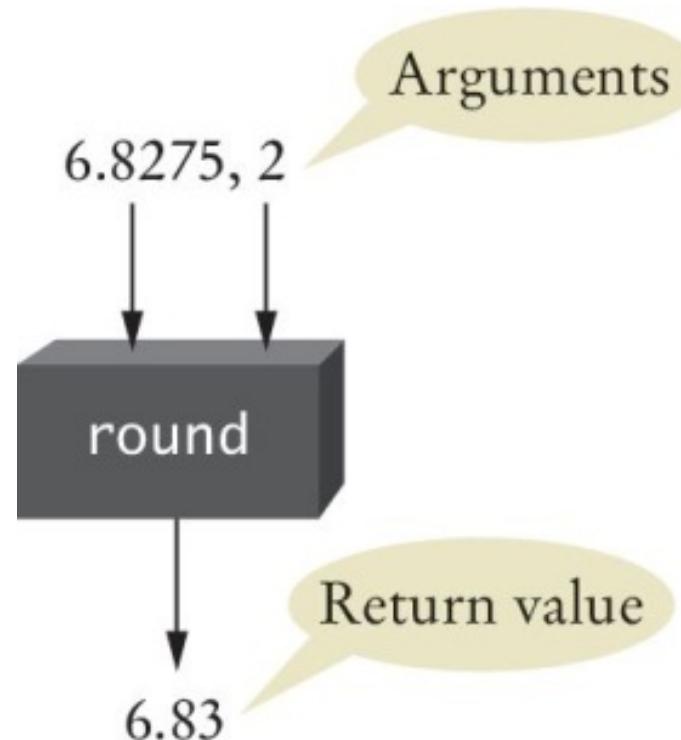
- A function is a sequence of instructions with a name
- For example, the round function, which was introduced in Chapter 2, contains instructions to round a floating-point value to a specified number of decimal places
- You *call* a function in order to execute its instructions

```
price = round(6.8275, 2) # Sets result to 6.83
```

- By using the expression `round(6.8275, 2)`, your program *calls* the round function, asking it to round 6.8275 to two decimal digits
- Functions can receive multiple arguments or it is also possible to have functions with no arguments

The `round` Function as a Black Box

- You pass the `round` function its necessary arguments (`6.8275 & 2`) and it produces its result (`6.83`)



Function Return Values

- The “output” that the round function computes is called the **return value**
- Functions return only one value
- The return value of a function is returned to the point in your program where the function was called

```
price = round(6.8275, 2)
```

- When the round function returns its result, the return value is stored in the variable ‘price’ statement)

Function Return Values

- Do not confuse returning a value with producing program output which is produced when using a `print()` statement

```
str1 = "pilot"  
str2 = "PILOT"
```

```
def a_function ():  
    """ this function returns False """  
    return (str1 == str2)
```

```
print(a_function_that_returns())
```

Designing Your Own Functions

- When you design your own functions, you will want to make them appear as black boxes to other programmers
 - Even if you are the only person working on a program, making each function into a black box pays off: there are fewer details that you need to keep in mind

Implementing and Testing Functions

SECTION 5.2

The "Design Recipe" for Functions

1. What's the **input** and where does it come from?
2. What's the **output** and where does it go?
3. Write a **contract** that states the function's name and the names and types of its parameters and return value.

Restaurant_price: rest: Restaurant -> float

or it can use actual Python annotations in the actual Python function header (also called the function signature):

def Restaurant_price(rest: Restaurant) -> float:

4. Write a one-line **purpose statement** or **docstring** that says in simple English what the function does.

" Take a total restaurant bill and compute the tip on that specified amount "

5. Write some **examples** of calling the function

```
assert triple(0) == 0
```

```
assert triple(5) == 15
```

6. Write the **body** of the function.

7. Run the function (which runs the **tests**—your examples from step 5) and check the results.

Implementing and Testing Functions

- A function to calculate the volume of a cube
 - What does it need to do its job?
 - What does it answer with?
- When writing ('defining') this function
 - Pick a name for the function (**cube**)
 - Declare a variable for each incoming argument
(side_length) (called parameter variables)
 - Put all this information together along with the def keyword to form the first line of the function's definition:

```
def cube(side_length):
```

This line is called the **header** of the function

Testing a Function

- If you run a program containing just the function definition, then nothing happens
 - After all, nobody is calling the function
- In order to test the function, your program should contain
 - The definition of the function
 - Include Assert Statements that call the function to test the result is as you expect.

assert function_you_wrote() == “does what you expect”

Calling/Testing the Cube Function

Implementing the function (function definition)

```
def cube(side_length):  
    """ returns cubed input """  
    volume = side_length ** 3  
    return volume
```

Calling and testing the function

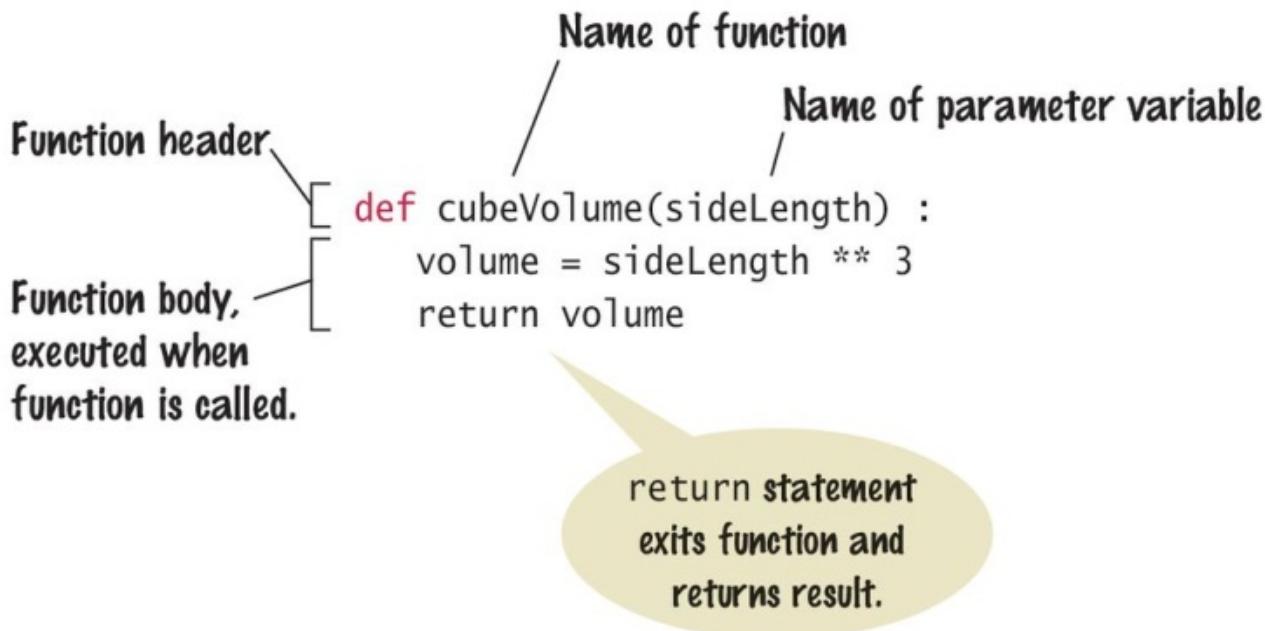
```
result1 = cube(2)  
result2 = cube(10)  
  
assert cube(2) == 8  
assert cube(10) == 1000
```

In your IDE

- Practice this thing :)

Syntax: Function Definition

Syntax `def functionName(parameterName1, parameterName2, . . .) :`
 statements



Return Values

SECTION 5.4

Return Values

- Functions can (optionally) return one value
 - Add a **return statement** that returns a value
 - A **return statement** does two things:
 - 1) Immediately terminates the function
 - 2) Passes the return value back to the calling function

```
def cube (side_length):  
    volume = side_length * 3  
    return volume
```

A yellow curly brace is positioned under the word "return" in the code, indicating its scope.

return

return statement

The return value may be a value, a variable or a calculation

Make Sure A Return Catches All Cases

- Missing return statement
 - Make sure all conditions are handled
 - In this case, `side_length` could be equal to 0
 - No return statement for this condition
 - The compiler will *not* complain if any branch has no return statement
 - It may result in a run-time error because Python returns the special value `None` when you forget to return a value

```
def cube (side_length) :  
    if side_length >= 0 :  
        return side_length ** 3  
    # Error--no return value if sideLength < 0
```

Make Sure A Return Catches All Cases

- A correct implementation:

```
def cube (side_length) :  
    result = 0  
    if side_length >= 0  
        result = side_length ** 3  
    return result
```

Functions Without Return Values

SECTION 5.5

Functions Without Return Values

- functions are not required to return a value
 - No return statement is required
 - The function can generate output even when it doesn't have a return value

```
...  
box_string("Hello")  
...
```

```
-----  
!Hello!  
-----
```

```
def box_string(contents:str) -> None:  
    n = len(contents) :  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

Using `return` Without a Value

- You can use the `return` statement without a value
 - The function will terminate immediately!

```
def boxString(contents) :  
    n = len(contents)  
    if n == 0 :  
        return # Return immediately  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

Reusable Functions

SECTION 5.6

Problem Solving: Reusable Functions

- Find repetitive code
 - May have different values but same logic

```
hours = int(input("Enter a value between 0 and 23: "))
while hours < 0 or hours > 23 :
    print("Error: value out of range.")
    hours = int(input("Enter a value between 0 and 23: "))
```

0 - 23

```
minutes = int(input("Enter a value between 0 and 59: "))
while minutes < 0 or minutes > 59 :
    print("Error: value out of range.")
    minutes = int(input("Enter a value between 0 and 59: "))
```

0 - 59

Write a ‘Parameterized’ Function

```
## Prompts a user to enter a value up to a given maximum until the user
provides
# a valid input.
# @param high an integer indicating the largest allowable input
# @return the integer value provided by the user (between 0 and high,
inclusive)
#
def readIntUpTo(high) :
    value = int(input("Enter a value between 0 and " + str(high) + ": "))
    while value < 0 or value > high :
        print("Error: value out of range.")
    value = int(input("Enter a value between 0 and " + str(high) + ": "))

    return value
```

Readtime.py

- Open the file readtime.py
- Test the program with several inputs
 - How would you modify your project to use the readInBetween function?

An Alternate If Structure

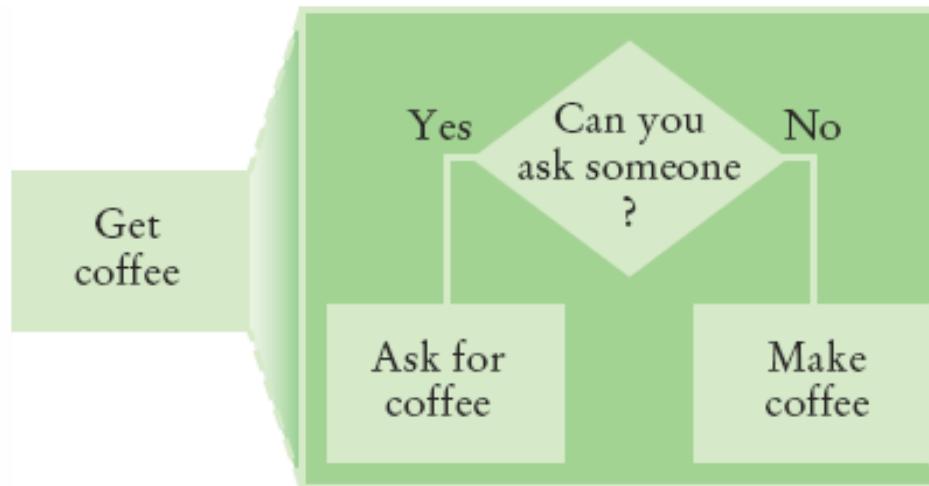
- Open the file `earthquake.py`
- The file contains two functions that solve the Richter scale problem from earlier this semester
 - The first uses an “`if - elif`” construct
 - The second uses single-line compound statements (Special Topic 5.1, p. 256)
 - This form of an `if` statement is very useful in functions that select and return a single value from a set of values

Stepwise Refinement

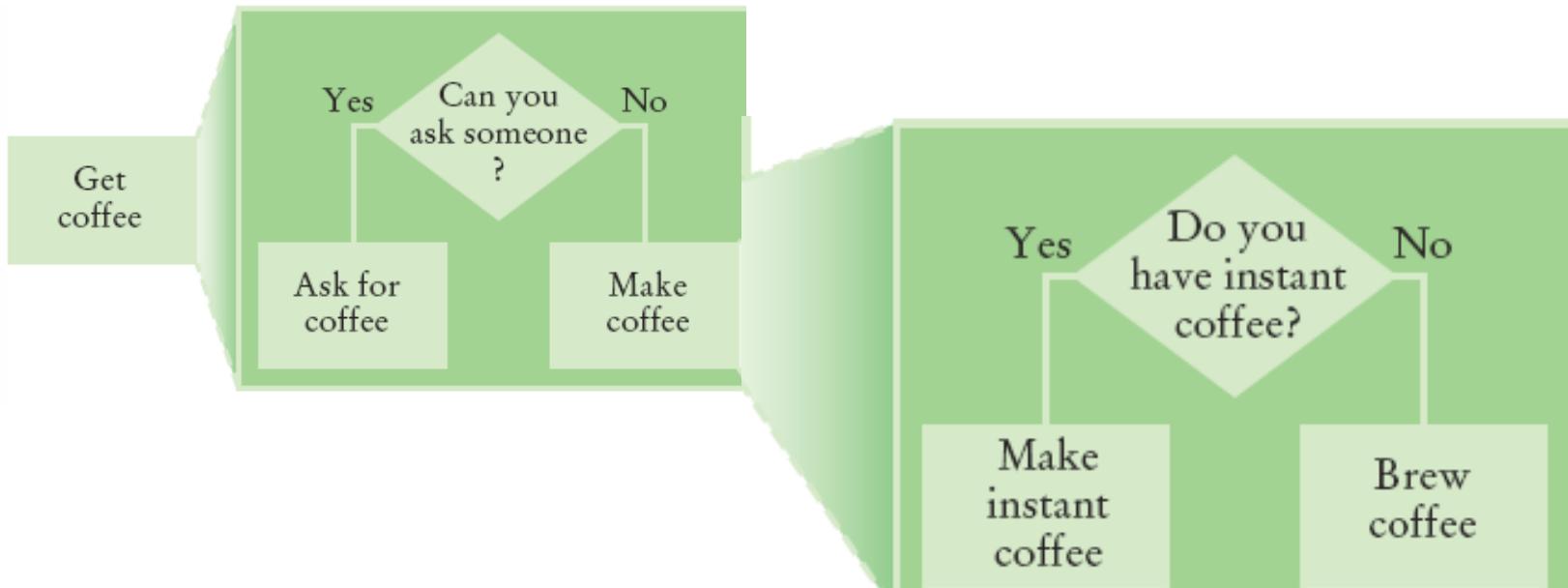
SECTION 5.7

Stepwise Refinement

- To solve a difficult task, break it down into simpler tasks
- Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve



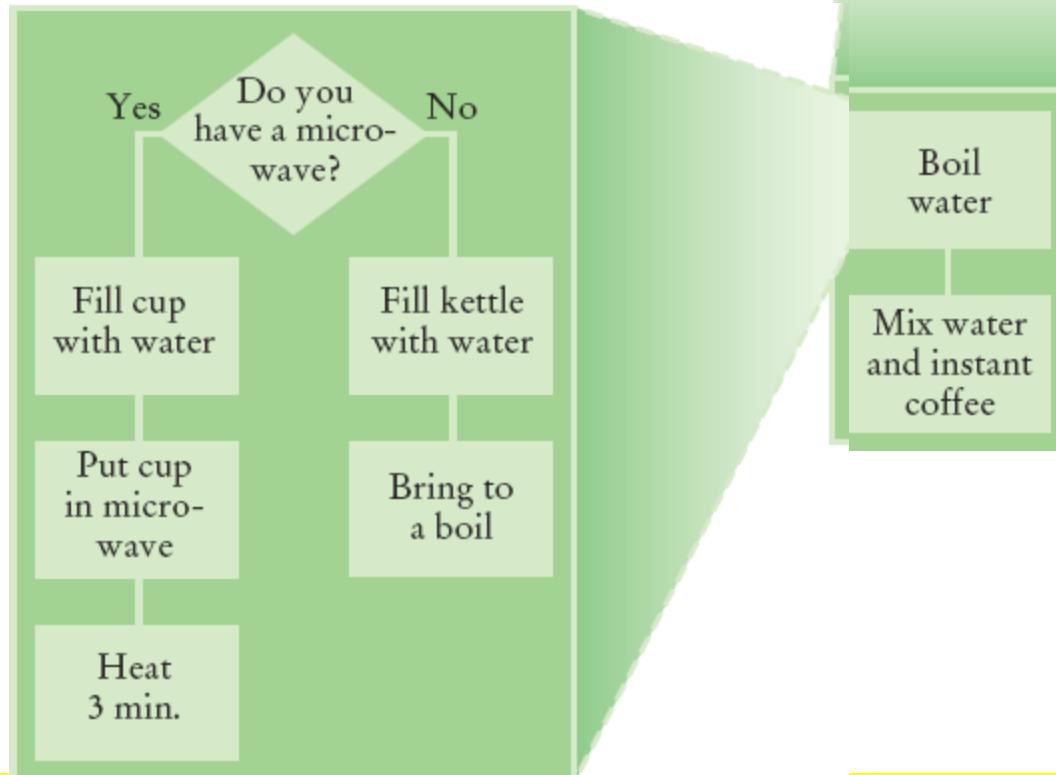
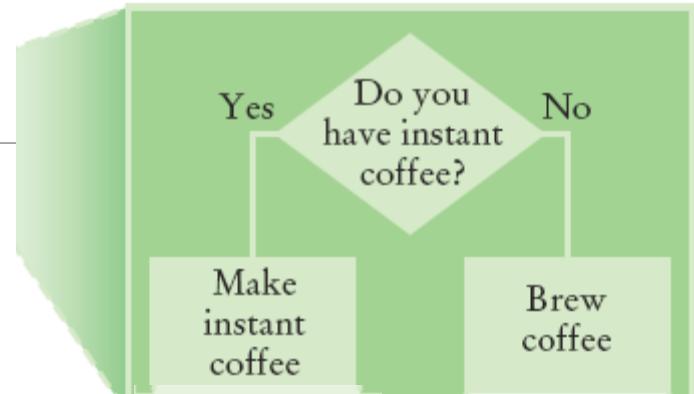
Get Coffee



- If you must make coffee, there are two ways:
 - Make Instant Coffee
 - Brew Coffee

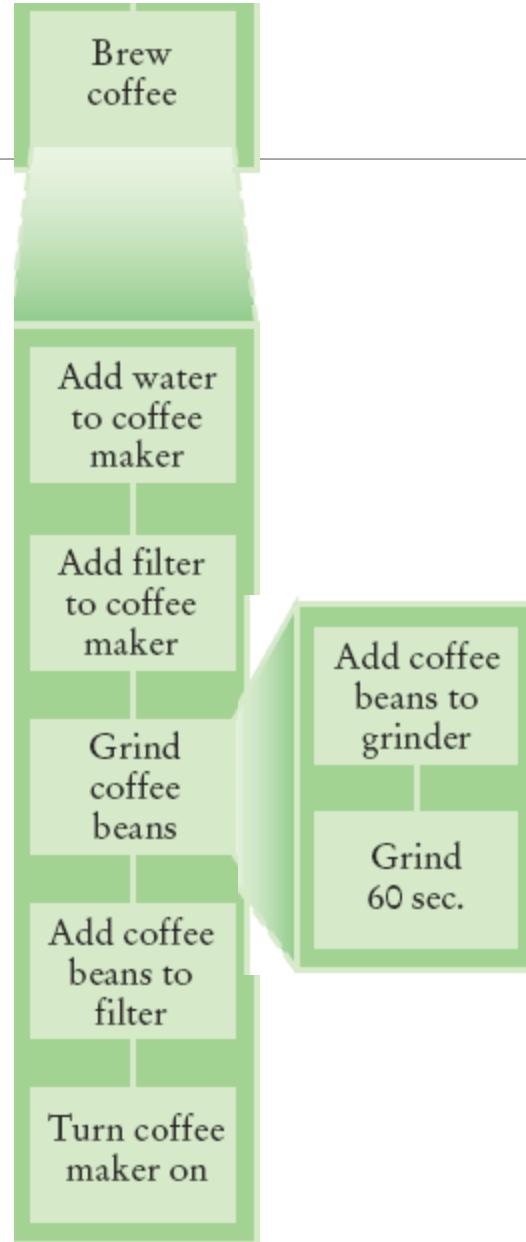
Instant Coffee

- Two ways to boil water
 - 1) Use Microwave
 - 2) Use Kettle on Stove



Brew Coffee

- Assumes coffee maker
 - Add water
 - Add filter
 - Grind Coffee
 - Add beans to grinder
 - Grind 60 seconds
 - Fill filter with ground coffee
 - Turn coffee maker on
- Steps are easily done



Stepwise Refinement Example

- When printing a check, it is customary to write the check amount both as a number (“\$274.15”) and as a text string (“two hundred seventy four dollars and 15 cents”)
- Write a program to turn a number into a text string
- Wow, sounds difficult!
- Break it down
 - Let’s take the dollar part (274) and come up with a plan
 - Take an Integer from 0 – 999
 - Return a String
 - Still pretty hard...

Stepwise Refinement Example

- Take it digit by digit (2, 7, 4) – left to right
- Handle the first digit (hundreds)
 - If empty, we are done with hundreds
 - Get first digit (Integer from 1 – 9)
 - Get digit name (“one”, “two”, “three”...)
 - Add the word “hundred”
 - Sounds easy!
- Second digit (tens)
 - Get second digit (Integer from 0 – 9)
 - If 0, we are done with tens... handle third digit
 - If 1, ... may be eleven, twelve... Teens... Not easy!
 - Let’s look at each possibility left (1x-9x)...

Stepwise Refinement Example

- If second digit is a 0
 - Get third digit (Integer from 0 – 9)
 - Get digit name ("", "one", "two"...) ... Same as before?
 - Sounds easy!
- If second digit is a 1
 - Get third digit (Integer from 0 – 9)
 - Return a String ("ten", "eleven", "twelve"...)
- If second digit is a 2-9
 - Start with string "twenty", "thirty", "forty"...
 - Get third digit (Integer from 0 – 9)
 - Get digit name ("", "one", "two"...) ... Same as before
 - Sounds easy!

Name the Sub-Tasks

- digitName
 - Takes an Integer from 0 – 9
 - Return a String (“”, “one”, “two”...)
- tensName (second digit ≥ 20)
 - Takes an Integer from 0 – 9
 - Return a String (“twenty”, “thirty”...) plus
 - digitName(third digit)
- teenName
 - Takes an Integer from 0 – 9
 - Return a String (“ten”, “eleven”...)

Write Pseudocode

part = number (The part that still needs to be converted)

name = "" (The name of the number)

If part >= 100

name = **name of hundreds in part** + " hundred"

Remove hundreds from part

If part >= 20

Append **tensName(part)** to name

Remove tens from part

Else if part >= 10

Append **teenName(part)** to name

part = 0

If (part > 0)

Append **digitName(part)** to name

***Identify functions that we can use
(or re-use!) to do the work***

Plan The Functions

- Decide on name, parameter(s) and types and return type
- def intName (number):
 - Turns a number into its English name
 - Returns a String that is the English description of a number (e.g., “seven hundred twenty nine”)
- def digitName (digit):
 - Return a String (“”, “one”, “two”...)
- def tensName (number):
 - Return a String (“twenty”, “thirty”...) plus
 - Return from digitName(thirdDigit)
- def teenName (number):
 - Return a String (“ten”, “eleven”...)

Convert to Python: intName Function

- Open the file intname.py in Wing
- main calls intName
 - Does all the work
 - Returns a String
- Uses functions:
 - tensName
 - teenName

```
5 def main() :  
6     value = int(input("Please enter a positive integer < 1000: "))  
7     print(intName(value))  
8
```

intName

```
13 def intName(number) :
14     part = number    # The part that still needs to be converted.
15     name = ""      # The name of the number.
16
17     if part >= 100 :
18         name = digitName(part // 100) + " hundred"
19         part = part % 100
20
21     if part >= 20 :
22         name = name + " " + tensName(part)
23         part = part % 10
24     elif part >= 10 :
25         name = name + " " + teenName(part)
26         part = 0
27
28     if part > 0 :
29         name = name + " " + digitName(part)
30
31 return name
```

digitName

```
37 def digitName(digit) :  
38     if digit == 1 : return "one"  
39     if digit == 2 : return "two"  
40     if digit == 3 : return "three"  
41     if digit == 4 : return "four"  
42     if digit == 5 : return "five"  
43     if digit == 6 : return "six"  
44     if digit == 7 : return "seven"  
45     if digit == 8 : return "eight"  
46     if digit == 9 : return "nine"  
47     return ""
```

teenName

```
53 def teenName(number) :  
54     if number == 10 : return "ten"  
55     if number == 11 : return "eleven"  
56     if number == 12 : return "twelve"  
57     if number == 13 : return "thirteen"  
58     if number == 14 : return "fourteen"  
59     if number == 15 : return "fifteen"  
60     if number == 16 : return "sixteen"  
61     if number == 17 : return "seventeen"  
62     if number == 18 : return "eighteen"  
63     if number == 19 : return "nineteen"  
64     return ""
```

tensName

```
70 def tensName(number) :
71     if number >= 90 : return "ninety"
72     if number >= 80 : return "eighty"
73     if number >= 70 : return "seventy"
74     if number >= 60 : return "sixty"
75     if number >= 50 : return "fifty"
76     if number >= 40 : return "forty"
77     if number >= 30 : return "thirty"
78     if number >= 20 : return "twenty"
79     return ""
```

Programming Tips

- Keep functions short
 - If more than one screen, break into ‘sub’ functions
- Trace your functions
 - One line for each step
 - Columns for key variables
- Use Stubs as you write larger programs
 - Unfinished functions that return a ‘dummy’ value

<i>intName(number=416)</i>	
<i>part</i>	<i>name</i>
416	""
16	"farhundee"
0	<i>"far hundred sixteen"</i>

Variable Scope

SECTION 5.8

Variable Scope

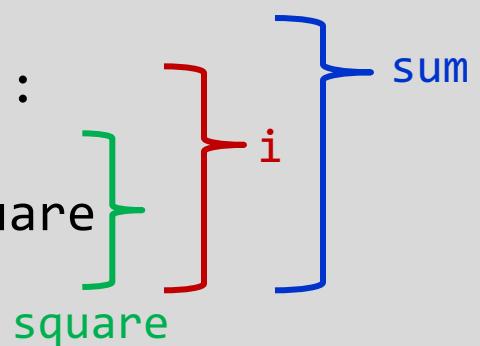
- Variables can be declared:
 - Inside a function
 - Known as ‘local variables’
 - Only available inside this function
 - Parameter variables are like local variables
 - Outside of a function
 - Sometimes called ‘global scope’
 - Can be used (and changed) by code in any function
- How do you choose?

The scope of a variable is the part of the program in which it is visible

Examples of Scope

- `sum`, `square` & `i` are local variables in `main`

```
def main() :  
    sum = 0  
        for i in range(11) :  
            square = i * i  
            sum = sum + square  
    print(square, sum)
```



Local Variables of functions

- Variables declared inside one function are not visible to other functions
 - `sideLength` is local to main
 - Using it outside main will cause a compiler error

```
def main():
```

```
    sideLength = 10
```

```
    result = cubeVolume()
```

```
    print(result)
```

```
def cubeVolume():
```

```
    return sideLength * sideLength * sideLength # ERROR
```

Re-using Names for Local Variables

- Variables declared inside one function are not visible to other functions
 - `result` is local to `square` and `result` is local to `main`
 - They are two different variables and do not overlap
 - This can be very confusing

```
def square(n):  
    result = n * n  
    return result
```

} `result`

```
def main():  
    result = square(3) + square(4)  
    print(result)
```

} `result`

Global Variables

- They are variables that are defined outside functions
- A global variable is visible to all functions that are defined after it
- However, any function that wishes to use a global variable must include a global declaration

Example Use of a Global Variable

- If you omit the global declaration, then the balance variable inside the withdraw function is considered a local variable

```
balance = 10000    # A global variable
def withdraw(amount) :
    # This function intends to access the
    # global 'balance' variable
    global balance
    if balance >= amount :
        balance = balance - amount
```

Programming Tip

- There are a few cases where global variables are required (such as `pi` defined in the `math` module), but they are quite rare
- Programs with global variables are difficult to maintain and extend because you can no longer view each function as a “black box” that simply receives arguments and returns a result
- Instead of using global variables, use function parameter variables and return values to transfer information from one part of a program to another

Graphics: Building an Image Processing Toolkit

SECTION 5.9

Building Your Own Tools

- As we learned earlier, Python's standard libraries contain a large collection of function and classes
 - You import a module (containing functions and / or classes) into your program if you need it
- What do you do when the standard libraries don't have what you need?
 - You create your own collection of tools and organize them into modules or a *software toolkit*

Creating Your Toolkit

- The tools or functions in the toolkit should all be related
 - The function names should be easy to remember
 - The parameters should be as consistent as possible
- Place the functions in a separate file
- Import them into your programs as needed

Recursive Functions

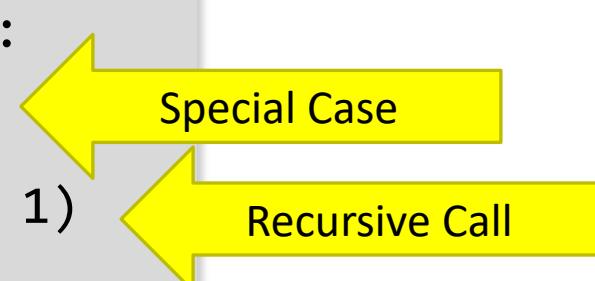
SECTION 5.10

Recursive Functions

- A recursive function is a function that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs

Recursive Triangle Example

```
def printTriangle(sideLength) :  
    if sideLength < 1 : return  
    printTriangle(sideLength - 1)  
    print("[]" * sideLength)
```



- The function will call itself (and not output anything) until `sideLength` becomes < 1
- It will then use the `return` statement and each of the previous iterations will print their results
 - 1, 2, 3 then 4

[]
[] []
[] [] []
[] [] [] []

Print the triangle with side length 3.
Print a line with four [].

Recursive Calls and Returns

Here is what happens when we print a triangle with side length 4.

- The call `printTriangle(4)` calls `printTriangle(3)`.
 - The call `printTriangle(3)` calls `printTriangle(2)`.
 - The call `printTriangle(2)` calls `printTriangle(1)`.
 - The call `printTriangle(1)` calls `printTriangle(0)`.
 - The call `printTriangle(0)` returns, doing nothing.
 - The call `printTriangle(1)` prints `[]`.
 - The call `printTriangle(2)` prints `[][]`.
 - The call `printTriangle(3)` prints `[][][]`.
 - The call `printTriangle(4)` prints `[][][][]`.

A Second Example

- Open the file digits.py
- This program computes the sum of the digits in a number (n)
 - We solved this last chapter in Section 4.2
 - We will use $n = 1729$ as our example
- Our algorithm was:
 - Remove the last digit by computing $n // 10$ and add the remainder to our total
 - To use recursion we can use the recursive function:
 - $\text{digitsum}(n // 10) + n \% 10$
 - Our special case is $n == 0$ to terminate the recursion

Summary

Summary: Functions

- A function is a named sequence of instructions
- Arguments are supplied when a function is called
- The return value is the result that the function computes
- When declaring a function, you provide a name for the function and a variable for each argument
- Function comments explain the purpose of the function, the meaning of the parameters and return value, as well as any special requirements
- Parameter variables hold the arguments supplied in the function call

Summary: Function Returns

- The **return** statement terminates a function call and yields the function result
 - Complete computations that can be reused into functions
- Use the process of stepwise refinement to decompose complex tasks into simpler ones
 - When you discover that you need a function, write a description of the parameter variables and return values
 - A function may require simpler functions to carry out its work

Summary: Scope

- The scope of a variable is the part of the program in which the variable is visible
 - Two local or parameter variables can have the same name, provided that their scopes do not overlap
 - You can use the same variable name within different functions since their scope does not overlap
 - Local variables declared inside one function are not visible to code inside other functions

Summary: Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
 - For recursion to terminate, there must be special cases for the simplest inputs
 - The key to finding a recursive solution is reducing the input to a simpler input for the same problem
 - When designing a recursive solution, do not worry about multiple nested calls
 - Simply focus on reducing a problem to a slightly simpler one