

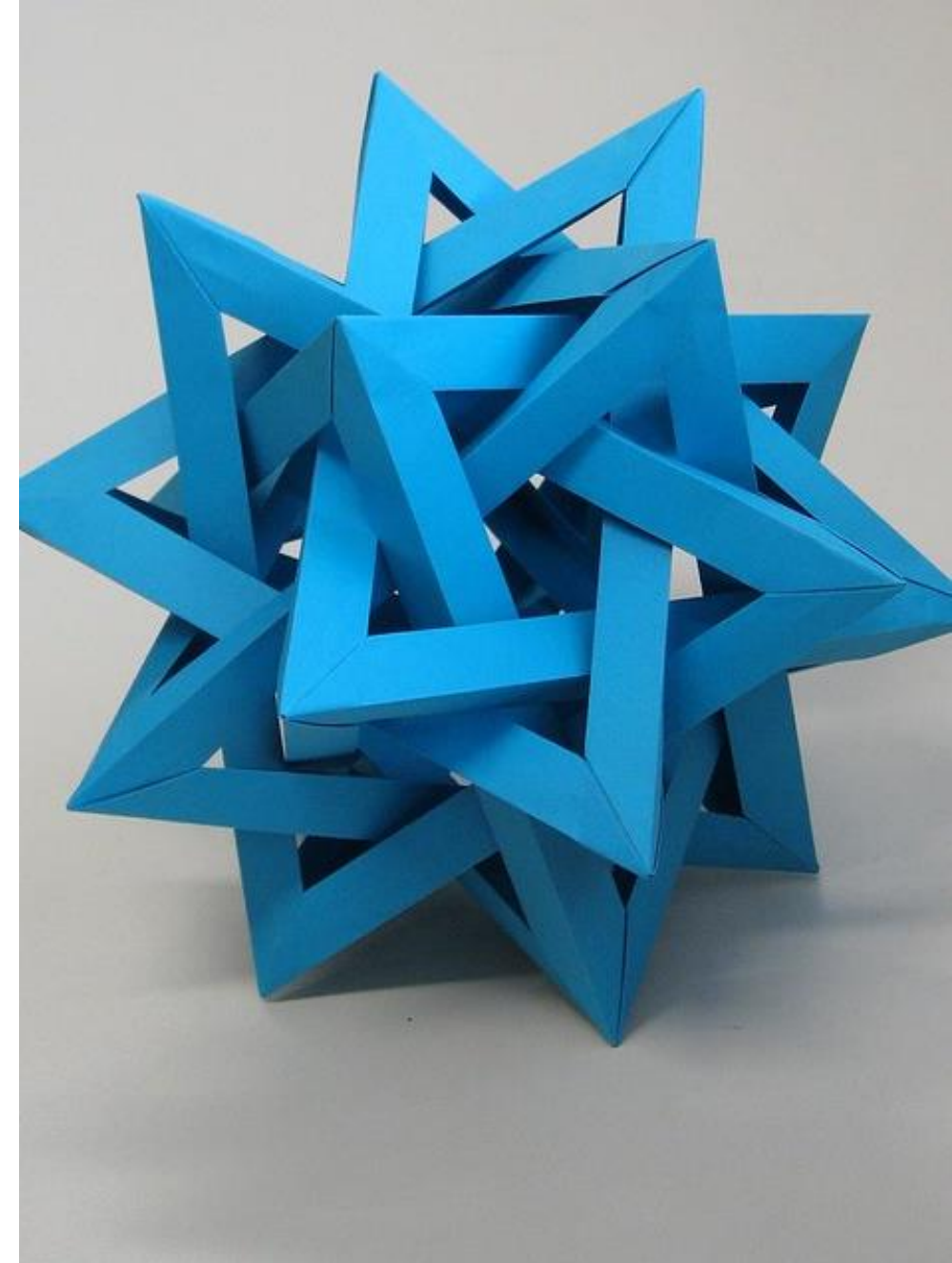


Unit P5: Functions

STRUCTURING YOUR CODE, FUNCTIONS,
PARAMETERS, RETURN VALUE



Chapter 5



Unit 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

*In this unit, 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

Functions as Black Boxes



5.1

Functions as Black Boxes

- A **function** is a sequence of **instructions** with a **name**
- For example, the **round** function contains instructions to round a floating-point value to a specified number of decimal places

```
round(number[, ndigits])
```

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are 0, and `round(1.5)` is 2). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise the return value has the same type as *number*.

<https://docs.python.org/3/library/functions.html#round>

Calling Functions

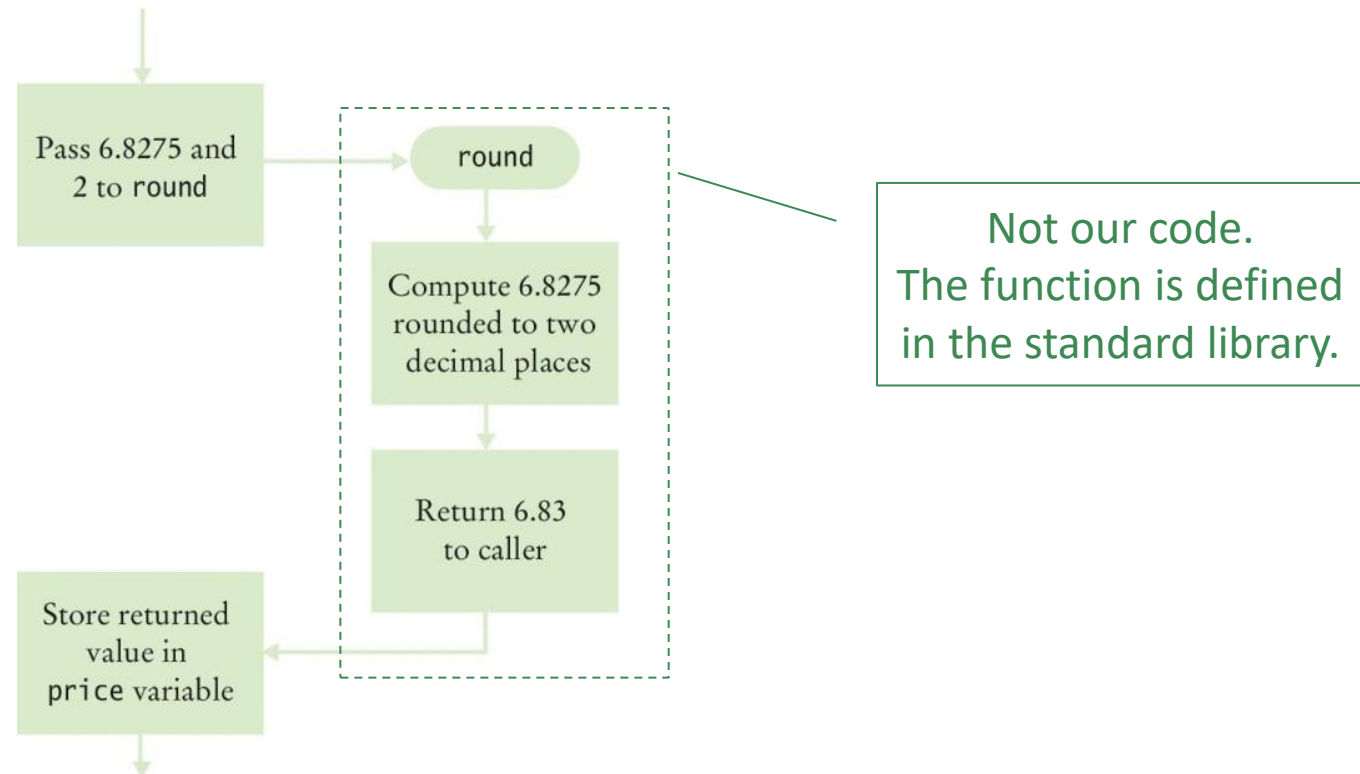
- You **call** a function in order to execute its instructions
`price = round(6.8275, 2) # Sets result to 6.83`
- By using the expression `round(6.8275, 2)`, your program **calls the round function**, asking it to round 6.8275 to two decimal digits

Calling Functions (2)

- 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
- When the function terminates, the computed **result is returned** by the function and may be **used** in an expression (e.g., assigned to `price`)
- After the value has been used, your program resumes execution

Calling Functions (3)

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



Function Arguments

- When another function calls the round function, **it provides “inputs”**, such as the values 6.8275 and 2 in the call `round(6.8275, 2)`
- These values are called the **arguments** of the function call
 - Note that they are not necessarily inputs provided by a human user
 - They are the values for which we want the function to compute a result
- Functions can receive multiple arguments
- It is also possible to have functions with no arguments

Function Return Values

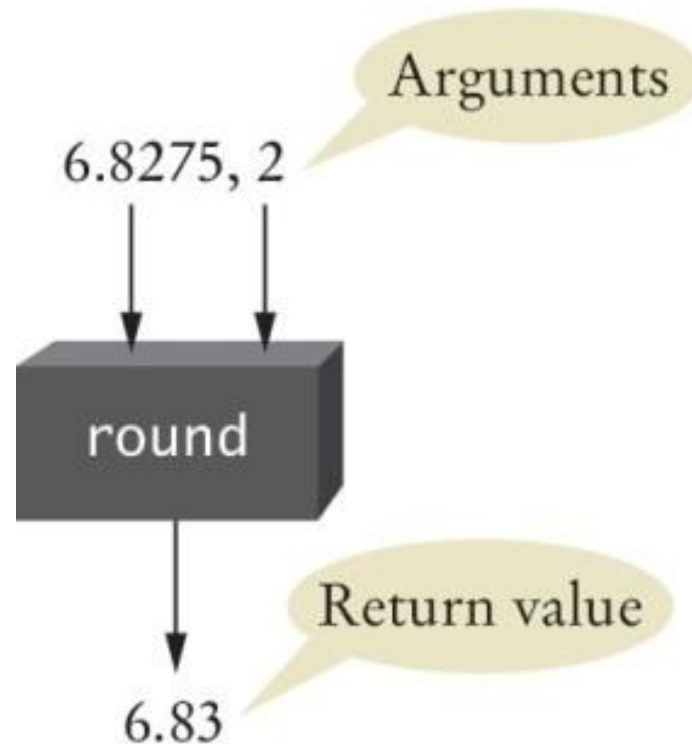
- The “**output**” that the round function computes is called **the return value**
- Functions return **only one value**
 - For multiple values, return a list or a tuple (see later...)
 - Some functions do not return any value
- The return value of a function is returned to the point in your program 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’

Black Box Analogy

- A thermostat is a 'black box'
 - Set a desired temperature
 - Turns on heater/AC as required
 - You don't have to know how it really works!
 - How does it know the current temp?
 - What signals/commands does it send to the heater or A/C?
- Use functions like 'black boxes'
 - Pass the function what it needs to do its job
 - Receive the answer

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)



The round Function as a Black Box

- You may wonder... how does the round function perform its job ?
- As a user of the function, you *don't need to know* how the function is implemented
- You just need to **know the specification** of the function:
 - If you provide arguments **x** and **n**, the function returns **x** rounded to **n** decimal digits
- **When you design your own functions, you will want to design them appear as black boxes**
 - Even if you are the only person working on a program, you want to use them as simple black boxes in the future, and let other programmers do the same

Where to Find Library Functions (1)

- Built-In functions in the Standard Library

- <https://docs.python.org/3/library/functions.html>

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Important
(already used)

useful
(have a look)

will use later
(with lists, dicts)

Where to Find Library Functions (2)

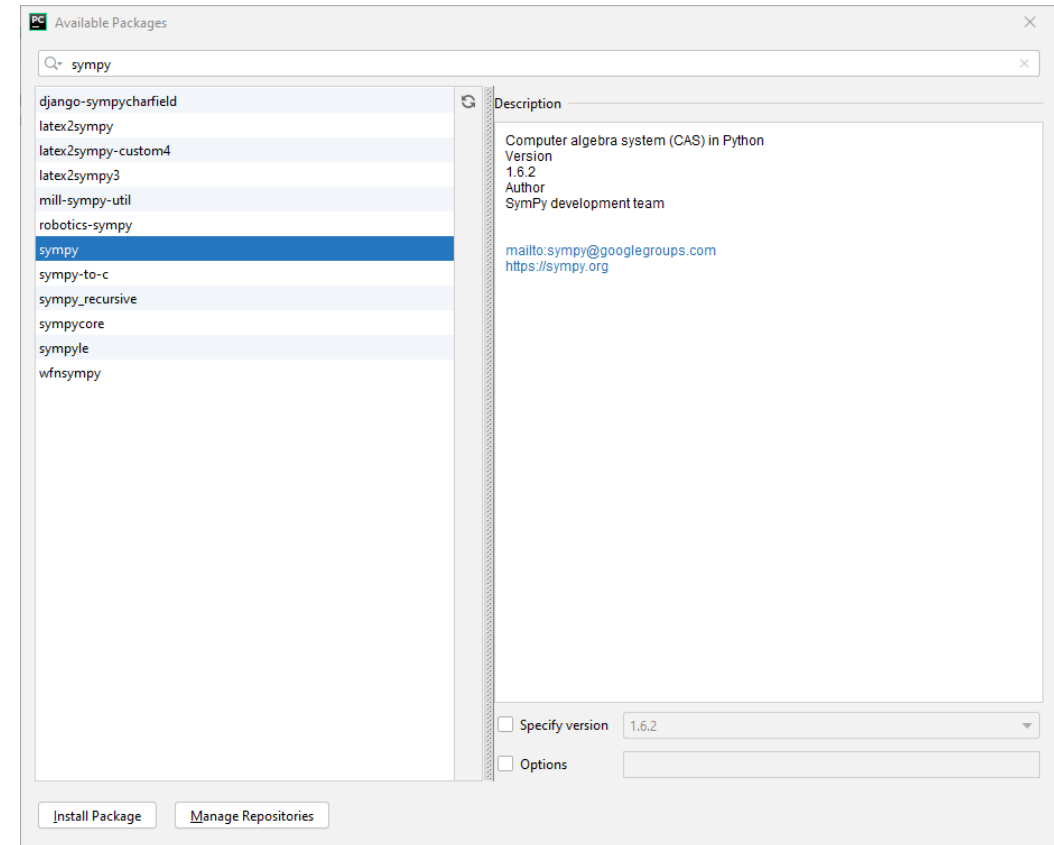
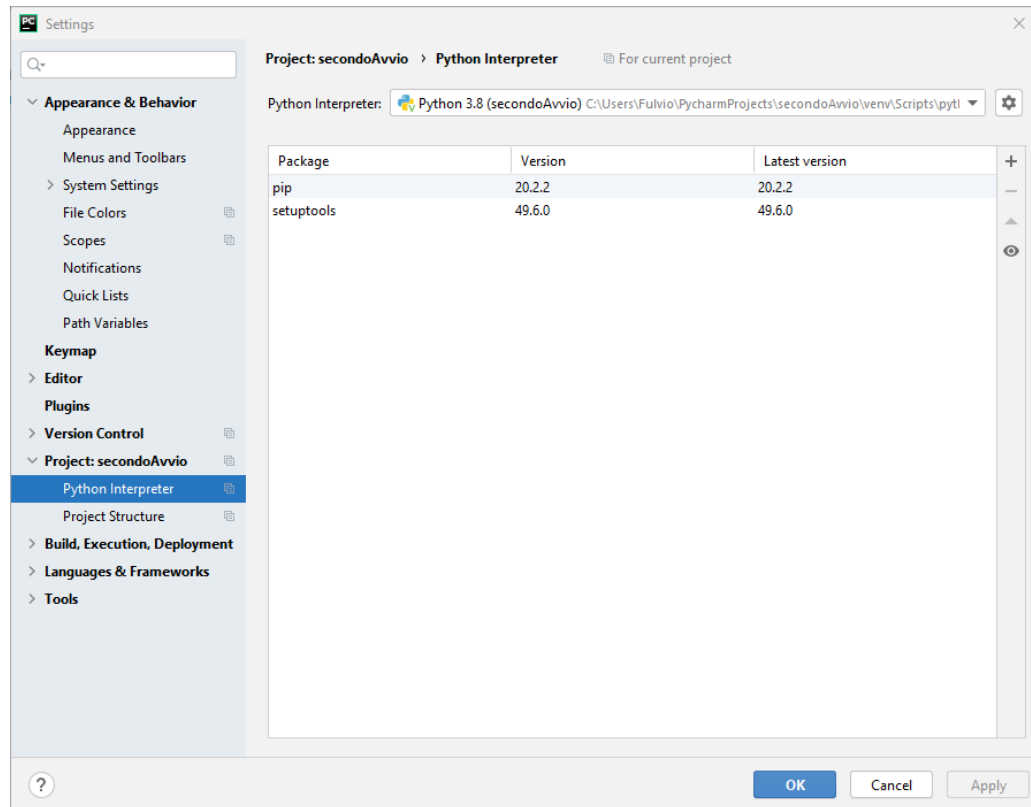
- Inside **Modules** in the Standard Library
 - <https://docs.python.org/3/library/>
 - <https://docs.python.org/3/py-modindex.html>
 - More than 200 modules, with many functions in each
 - Interesting ones: string, math, random, statistics, os.path, csv, json, ...
- Remember to `'import module'` or `'import function from module'`

Where to Find Library Functions (3)

- Available to **download** from the Python Package Index (**PyPI**) repository
 - <https://pypi.org/> - over 200k modules available
 - Install new modules in your project
 - with `'pip install module'`
 - from PyCharm Project Settings
 - ...
- Remember to `'import module'` or `'import function from module'`



Installing modules in PyCharm



Implementing and Testing Functions



5.2

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 (**cubeVolume**)
 - Declare a **variable** for each incoming **argument**
 - (**sideLength**) – list of parameter variables
 - Put all this information together along with the **def** keyword to form the first line of the function's definition:

```
def cubeVolume(sideLength):
```

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 (*function body*)
 - Should also compute the *return value*, with the return keyword
 - Statements that call the function and print the result

Calling/Testing the Cube Function

Implementing the function (function definition)

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

Calling/testing the function

```
result1 = cubeVolume(2)  
result2 = cubeVolume(10)  
print("A cube with side length 2 has volume", result1)  
print("A cube with side length 10 has volume", result2)
```

Syntax: Function Definition

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

The diagram shows a Python function definition with the following code and annotations:

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

Annotations:

- Name of function**: Points to `cubeVolume`.
- Name of parameter variable**: Points to `sideLength`.
- Function header**: Points to the entire first line `def cubeVolume(sideLength) :`.
- Function body, executed when function is called.**: Points to the indented lines `volume = sideLength ** 3` and `return volume`.

return statement
exits function and
returns result.

Programming Tip: Function Comments

- Whenever you write a function, you should **comment** its behavior
- Remember, comments are for human readers, not compilers

```
## Computes the volume of a cube.  
# @param sideLength the length of a side of the cube  
# @return the volume of the cube  
#  
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

Function comments explain the purpose of the function, the meaning of the parameter variables and the return value, as well as any special requirements

There are standards for automatically converting “structured” comments into documentation.

See: <https://realpython.com/documenting-python-code/>

The `main` Function

- When defining and using functions in Python, it is good programming practice to **place all statements into functions**, and to specify **one function as the starting point**
- Any legal name can be used for the starting point, but we chose '`main`' since it is the required function name used by other common languages
- Of course, we must have **one statement** in the program that **calls the `main` function**

Syntax: The `main` Function

By convention,
`main` is the starting point
of the program.

```
def main() :  
    result = cubeVolume(2)  
    print("A cube with side length 2 has volume", result)
```

The `cubeVolume`
function is defined below.

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

This statement is outside
any function definitions.

```
main()
```


Cubes.py with Documentation

```
1  ##
2  # This program computes the volumes of two cubes.
3  #
4
5  def main() :
6      result1 = cubeVolume(2)
7      result2 = cubeVolume(10)
8      print("A cube with side length 2 has volume", result1)
9      print("A cube with side length 10 has volume", result2)
10
11  ## Computes the volume of a cube.
12  # @param sideLength the length of a side of the cube
13  # @return the volume of the cube
14  #
15  def cubeVolume(sideLength) :
16      volume = sideLength ** 3
17      return volume
18
19  # Call the main function to begin executing the program.
20  main()
```

Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

Note

- In general, the call to the main function:
 - Should be executed if the program is executed directly, in standalone mode
 - Should not be executed if the program is imported, as a module, from a larger program
 - We should check the `__name__` special variable, that contains the name of the module (or `__main__` if standalone)
 - You often see this code:

```
if __name__ == '__main__':  
    # call the main function if we are running in  
    # standalone mode  
    # don't call it if we are imported as a module  
    main()
```



Sub-Note

- Many internal Python variables or functions have special names, and are not (should not) normally used
 - Novice programmers must avoid defining and using variable names starting with `'_'`
- To avoid confusion, system variables have a name with starting and ending double underscores
 - `__name__`
- They are called “dunder” names (for double-underscore)
 - `__name__` is read as *dunder*-name

Not needed in this
course

Using Functions: Order (1)

- It is important that you **define any function before you call it**
- For example, the following will produce a compile-time error:
`print(cubeVolume(10))`

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

- The compiler does not know that the cubeVolume function will be defined **later** in the program
 - Doesn't know what function to **call**

Using Functions: Order (2)

- However, a function can be called **from within another function** before the former has been defined

- The following is perfectly legal:

```
def main() :  
    result = cubeVolume(2) # 1  
    print("A cube with side length 2 has volume",  
          result)
```

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

```
main() # 2
```

- In #1, the function main is just **defined** (not yet **executed**). It will be called in #2, that is **after** the definition of cubeVolume.

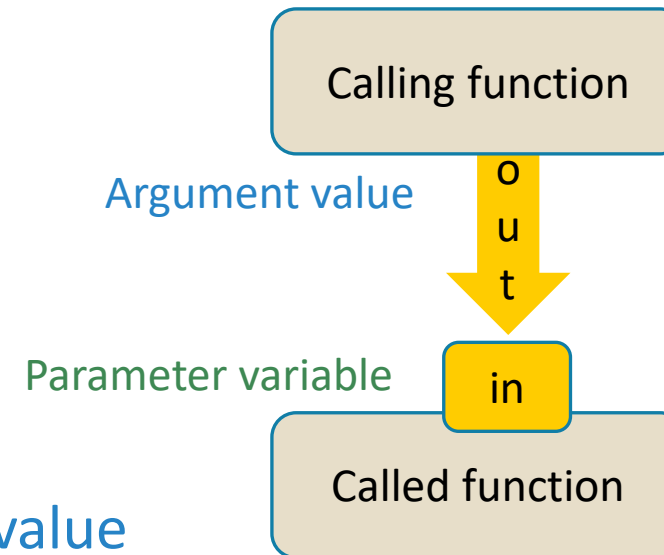
Parameter Passing



5.3

Parameter Passing

- **Parameter variables** receive the **argument values** supplied in the function call
- The **argument value** may be:
 - The current contents of a variable
 - A 'literal' value: 2, 3.14, 'hello'
 - Aka, '*actual parameter*' or **argument**
- The **parameter variable** is:
 - Declared in the called function
 - *Initialized* with the value of the **argument value**
 - *Used as a variable* inside the called function
 - Aka, '*formal parameter*'



Parameter Passing Steps

```
result1 = cubeVolume(2)
```

result1 = 8

```
def cubeVolume(sideLength):  
    volume = sideLength * 3  
    return volume
```

sideLength = 2

volume = 8

See it live on PythonTutor:

<http://pythontutor.com/live.html#mode=edit>

Common Error

- Trying to **modify parameter variables**
- A copy of the argument values is passed (the **value** is passed)
 - Called function (addTax) can modify local copy (price), only
 - The total variable in the caller function is **unaffected**
 - **total == 10** after the function call

```
total = 10  
addTax(total, 7.5);
```

Copy
value



total
10.0

```
def addTax(price, rate):  
    tax = price * rate / 100  
    # No effect outside the function  
    price = price + tax  
    return tax;
```

price
10.75

Programming Tip

- Do not modify parameter variables
- Many programmers find this practice confusing

```
def totalCents(dollars, cents) :  
    cents = dollars * 100 + cents # Modifies parameter variable.  
    return cents
```

- To avoid the confusion, simply introduce a separate variable:

```
def totalCents(dollars, cents) :  
    result = dollars * 100 + cents  
    return result
```

Return Values



5.4

Return Values

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

```
def cubeVolume (sideLength):  
    volume = sideLength * 3  
    return volume
```

return statement

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

Returning multiple values

- Only one value may be returned by a function
- If you need to return more than one value, you may return a tuple, containing the values
- Example:
 - `return (x, y)`
 - Build a tuple `(x, y)`
 - Return it

Returning no values

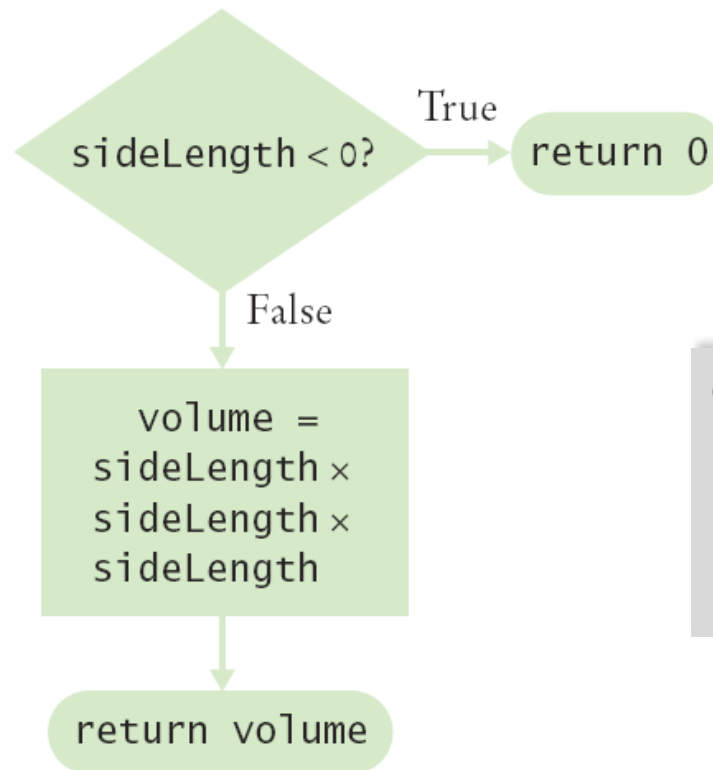
- If no return value is needed, you may simply return from a function without returning anything

`return # no value specified`

- If the return keyword is not encountered during the execution of a function, it is equivalent to having an `empty return after the last statement` of the function

Multiple `return` Statements

- A function can use **multiple return** statements
 - But **every branch** should lead the function to encounter a return statement



```
def cubeVolume(sideLength):  
    if (sideLength < 0):  
        return 0  
    return sideLength * 3
```

Multiple `return` Statements (2)

- Alternative to multiple returns (e.g., one for each branch):
 - You can avoid multiple returns by
 - Storing the function result in a variable
 - Returning the variable's value in the last statement of the function
 - For example:

```
def cubeVolume(sideLength) :  
    if sideLength >= 0:  
        volume = sideLength ** 3  
    else :  
        volume = 0  
    return volume
```


Make Sure a Return Catches All Cases

- Missing return statement
 - Make sure *all conditions* are handled
 - In this case, `sideLength` could be less than 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 cubeVolume(sideLength) :  
    if sideLength >= 0 :  
        return sideLength ** 3  
    # Error—no return value if sideLength < 0
```

Make Sure A Return Catches All Cases (2)

- A correct implementation:

```
def cubeVolume(sideLength) :  
    if sideLength >= 0 :  
        return sideLength ** 3  
    else :  
        return 0
```

Implementing a Function: Steps

- Describe what the function should do
 - Provide a simple “liberal arts terms” description of what the functions does
 - “Compute the volume of a pyramid with a square base”
- Determine a list of all of the function’s inputs
 - Make a list of all of the parameters that can vary
 - Do not be overly specific
- Determine the types of the parameter variables and the return value

Implementing a Function: Steps (2)

- Write pseudocode for obtaining the desired result
 - Express as mathematical formulas, branches and loops in pseudocode
- Implement the function body

```
def pyramidVolume(height, baseLength) :  
    baseArea = baseLength * baseLength  
    return height * baseArea / 3
```

Implementing a Function: Steps (3)

- Test your function
 - Design test cases and code

```
Volume: 300  
Expected: 300  
Volume: 0  
Expected: 0
```

Pyramids.py

- Open the file `pyramids.py`
- Look at how the main function is set up to make the calls to `pyramidVolume` and print the expected results

Functions Without Return Values



5.5

Functions Without Return Values

- Functions are **not required to return a value**
 - No return statement is required
 - The function can generate output (e.g., printing) even when it doesn't have a return value

```
...  
boxString("Hello")  
...
```

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

```
def boxString(contents) :  
    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



5.6

Problem Solving: Reusable Functions

- Find repetitive code

- May have different values but same logic

0 - 23

```
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 - 59

```
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: "))
```

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 readlnBetween function?

Variable Scope



5.8

Variable Scope

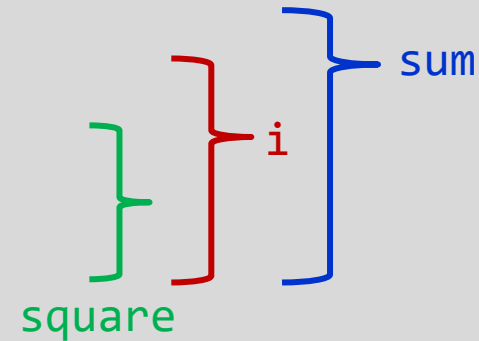
- Variables can be declared:
 - Inside a function
 - Known as 'local variables'
 - Only available inside this function
 - Parameter variables are like local variables
 - Outside of a function
 - Sometimes called 'global scope'
 - Can be used (and changed) by code in any function
- How do you choose?

*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

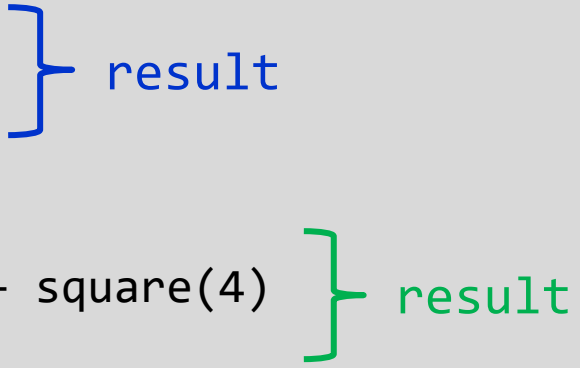
- Variables declared inside one function are not visible to other functions
 - `sideLength` is local to main
 - Using it outside main will cause a compiler error

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



Global Variables

- They are variables that are defined outside functions
- A global variable is visible to all functions
- However, any function that wishes to change 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

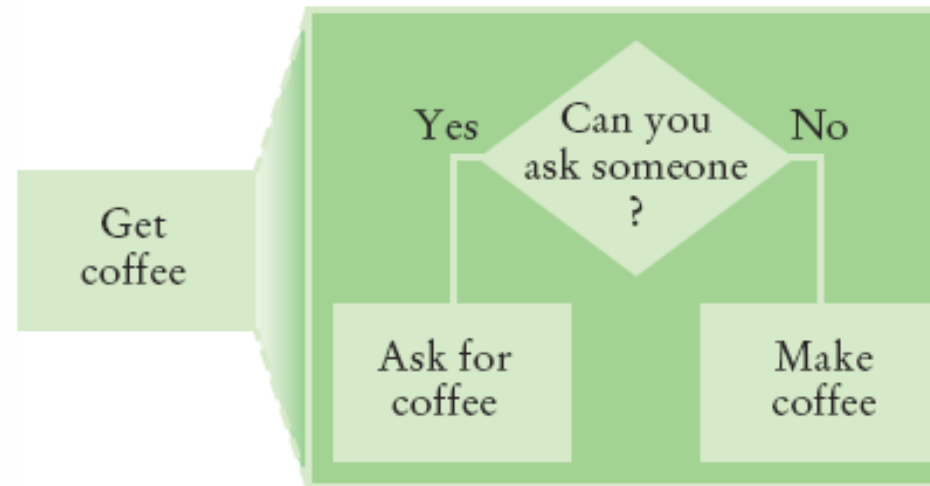
Stepwise Refinement



5.7

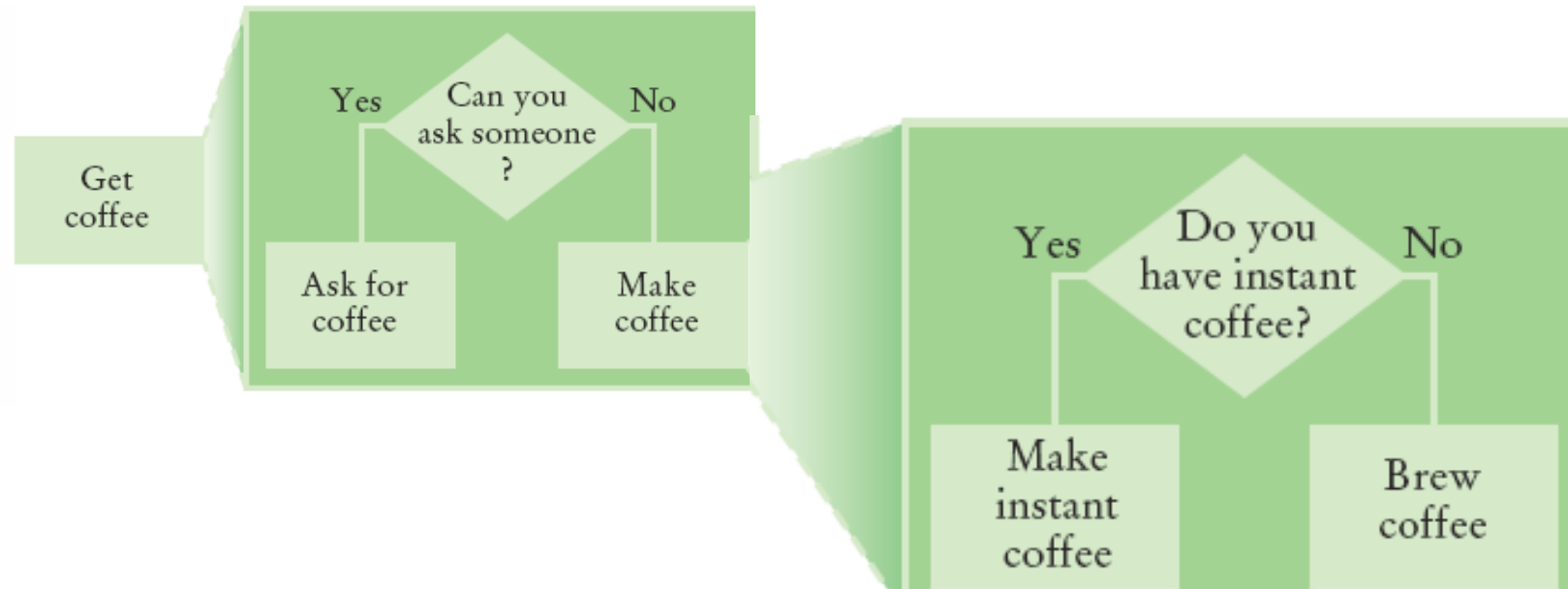
Stepwise Refinement

- 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



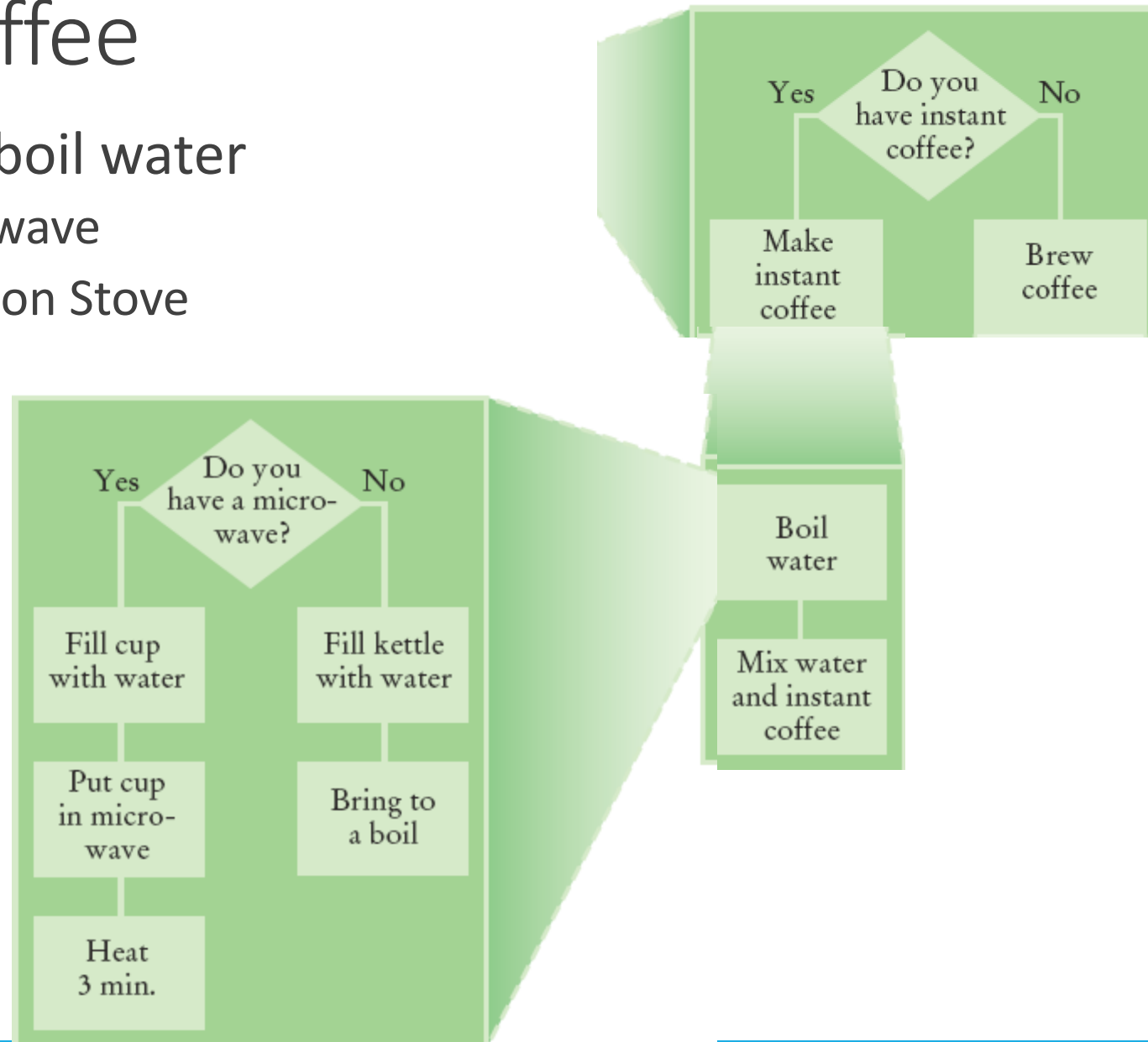
Get Coffee

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



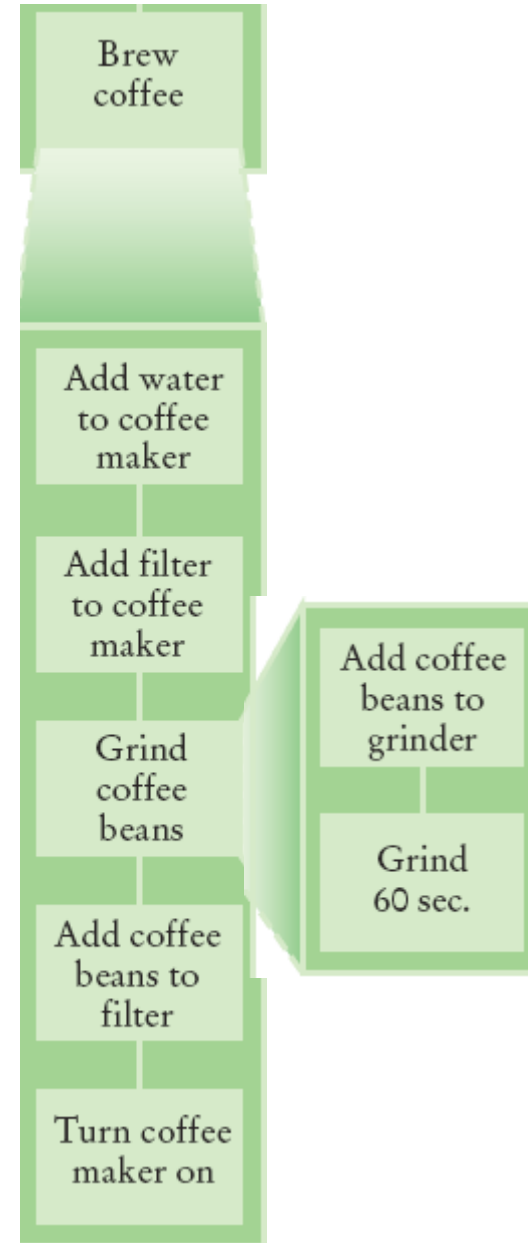
Instant Coffee

- 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 \geq 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
 - digitName

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

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  
  
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	"four hundred"
0	"four hundred sixteen"

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
- 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