Writing Functions

- What is a function?
- Function declaration and parameter passing
- Return values
- Objects as parameters
- Abstraction
- Data scoping
- Encapsulation
- Modules
- Recursion
- Other topics

- **Reading:**
  - *Dawson, Chapter 6*
  - http://introcs.cs.princeton.edu/python/21function/
  - http://introcs.cs.princeton.edu/python/22module/
  - http://introcs.cs.princeton.edu/python/23recursion

Organizing Code

- We have recently discussed the topic of organizing data in order to make it more manageable
- Similarly, you can also organize your *code* into logical, related units
- As you write code, you may find yourself frequently repeating a sequence of statements in order to accomplish a task
- In such cases, you will likely want to make those statements into a ***function***.

Why Functions?

- With simpler programs, separating groups of statements by white space may be enough
- However, as programs become more complex, numerous lines will be increasingly difficult to read, understand, and maintain.
- Also, it may become tedious to repeatedly type the same several lines of code.
- Creating functions allows you to make your code more ***organized*** and ***concise***.

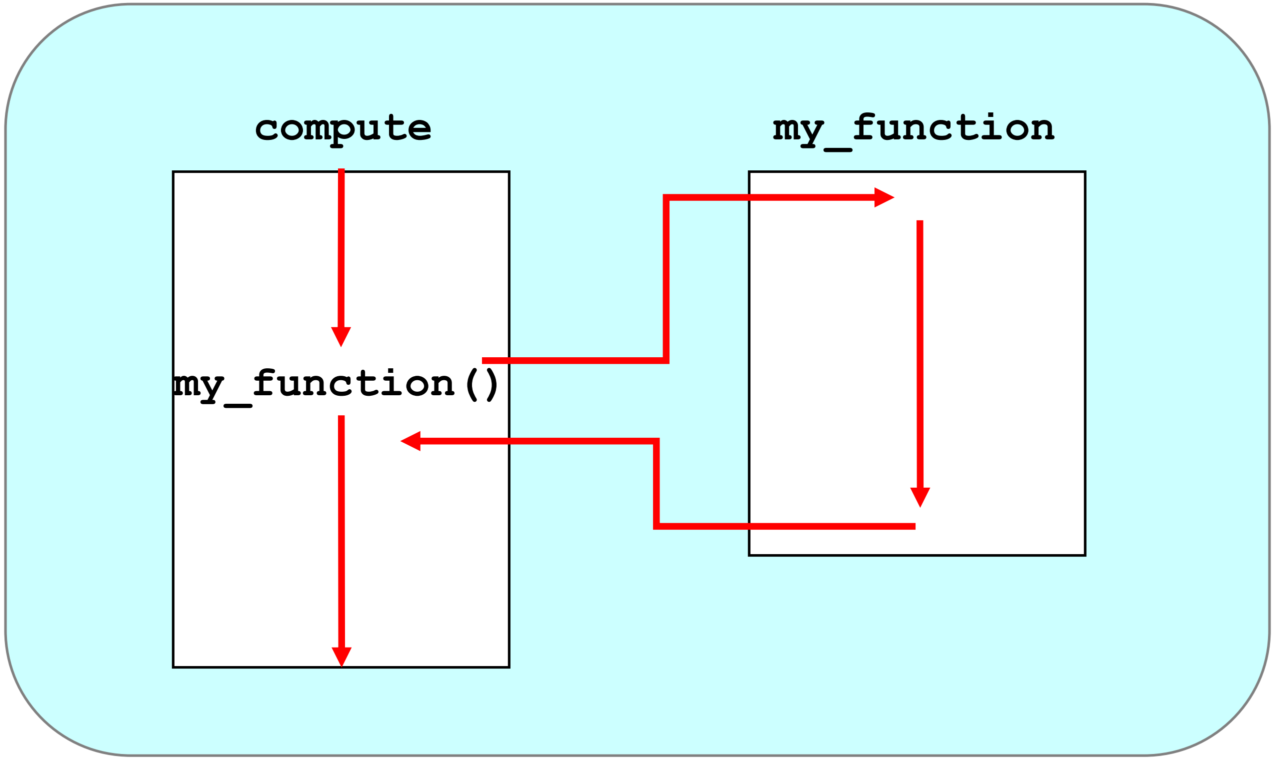
See **instructions.py**

What Is a Function?

- At the most basic level, a **function** is a *named block of code* that *accomplishes a task*
- When a function is *invoked*, the flow of control jumps to the function and executes its code
- When complete, the flow *returns* to the place where the function was called and continues
- The invocation may or may not *return a value*, depending on how the function is defined

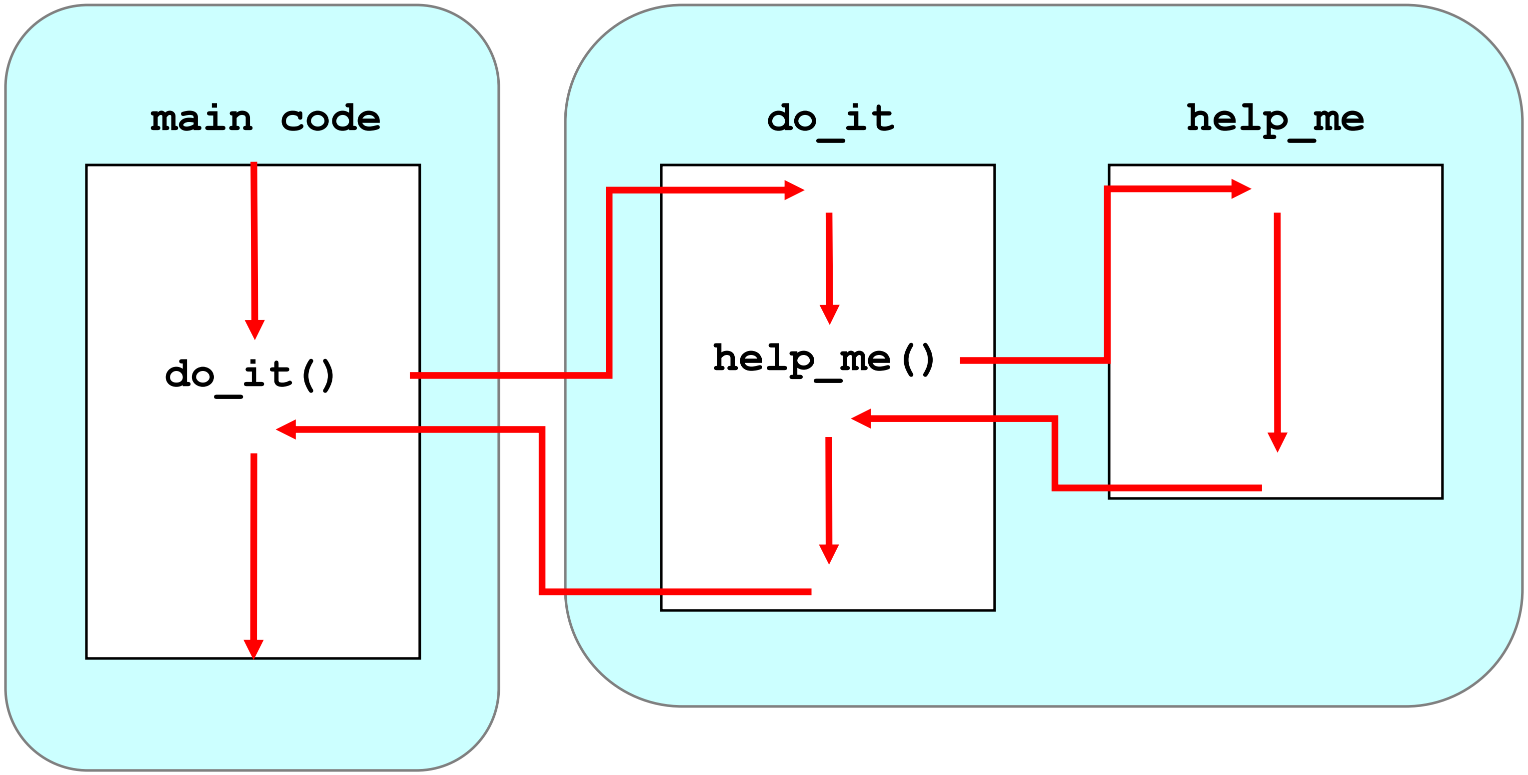
Function Control Flow

If the called function is a Python built-in (or in the same code file), then likely only the function name is needed



Function Control Flow

- The called function may, in fact, call another function



Creating Functions

- A *function definition* specifies the code that will be executed when the function is invoked (or "called").
- This definition has several components, some mandatory and some optional.
- At the very least, a function definition ***must*** have:
  1. A header
  2. A body
- With these requirements, there are variations

Function Header

- A function definition begins with a *function header*

```
def calc (num1, num2, message):
```

reserved word "def"

function name

parameter list

a colon : symbol

The parameter list specifies the name of each parameter

The name of a parameter in the function declaration and code is called a *formal parameter*

Function Body

- The function header is followed by the *function body*

```
def calc (num1, num2, message):
    sum = num1 + num2
    result = message[sum]
    return result
```

Be sure there is no conflict between your code and the parameter types

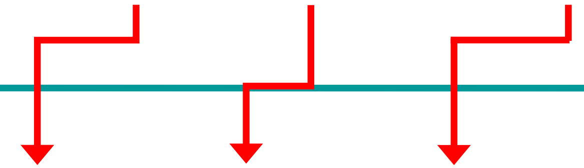
Make sure you know what type of data your function returns when called

- `sum` **and** `result` **are local data**
- **They are *created* each time the function is called, and are *destroyed* when it finishes executing**

Parameters

- When a function is called, the *actual parameters* in the call are copied into the *formal parameters* in the function header

```
ch = calc (25, count, "Hello")
```



```
def calc (num1, num2, message):
    sum = num1 + num2
    result = message[sum]
    return result
```

Objects as Parameters

- Another important issue related to function design involves parameter passing
- Since all data in Python are objects, that means parameters in a Python function are *passed by reference*
- When an object is passed to a function, the actual parameter and the formal parameter become **aliases** of each other – referring to *the same object*!
- For this reason, depending on the type of object, the function might change the object somehow.

Passing Objects to functions

- What a function does with a parameter may or may not have a permanent effect on the object. Ex.: die_changer.py

- **See also:**
    - parameter_tester.py
    - parameter_modifier.py
    - num.py

- Note the difference between changing the *internal state* of an object versus changing the value of a reference to point to a different object

The return Statement

- In addition to carrying out a set of instructions, a function may also return a value
- In other words, a function call may have a value that can be used like any other value in an expression.
- A *return statement* specifies the ***value*** that will be returned upon completion of the function

`return` *`expression`*

- You must be aware of the possible return types
- Recall `present_scores` and `final_scores`

See **receive and return.py**

Abstraction

- Functions bring up an important idea in programming: ***abstraction***
- “Abstraction” refers to the idea of focusing on the general idea about something, rather than the specific details.
- For example, if you order food at a restaurant, you know how to place the order and receive your food…
- …but you probably do not know what is specifically happening, in the restaurant, behind the scenes.

Abstraction

- Functions also exemplify the idea of abstraction.
- For example, consider the following code:

```
>>> my_list = [5, 4, 3, 2, 1]
>>> my_list.sort()
>>> print (my_list)
[1, 2, 3, 4, 5]
```

- There are many ways to sort a sequence. When we call the **sort** function for **my_list**, we do not actually know what algorithm the **sort** function is using.
- All we know is its result: the list items being sorted
- That is abstraction: We have an *outside* view but do not know (or care) what happens *internally*

Data Scoping

- Recall that a variable is a *named location in memory*. It does not exist until *declared*
- If you try to use a name for a variable that *does not exist*, then you will get an error. At that particular point in your program, the name *has no meaning* to the interpreter
- This relates not only to *whether* the variable has been declared but also to *where* it has been declared.

Data Scoping

- Different parts of your program, that are considered separate from one another, are called ***scopes***.
- Scopes in a program are of varying levels and degrees – from wider to narrower.
- A variable or function name – along with other identifiers – is only meaningful when created/used within a scope.
- Here, two levels of scope are of interest to us.

Global Data

- So far, most of our work in programs has been in the ***global*** scope – the highest level of our code file.

- Consider these lines in a code file:

```
my_list = ["Joe", "Sue", "Bill"]
list_len = len(my_list)
print ("My list:", my_list)
print ("Length:", list_len)
```

- The variables `my_list` and `list_len` would be global because you could use them at any level of the code, from that point forward.

Local Data

- Inside of a function's body, you have what is called the ***local*** scope. It consists of:
  1. The function's formal parameters
  2. Any variables declared inside the function
- Here, for example…

```
def calc (num1, num2, messge):
    sum = num1 + num2
    result = message[sum]
    return result
```

- …the variables **`num1`**, **`num2`**, **`messages`**, **`sum`**, and **`result`** are all local variables.

Local and Global Data

- When a function's code finishes, all local variables are destroyed (including the formal parameters) – and recreated the next time
- Previously created global variables may be used in the local scope, but local variables may only be used *in their own scope*.
- If you try to assign a value to a global variable within a local scope, you will actually be creating a new local variable (by the same name) *without affecting the global variable*

Local and Global Data

- This is called *shadowing* because – within the local scope – the local variable is now "hiding" the global variable of the same name.
- The global variable's value will not change
- The only exception is if you use the reserved word **`global`** in order to claim full access to it
- See, for example, **global_reach.py**
- It is important to know when to use global data

Encapsulation

- This, in turn, brings us to a topic important in many branches of programming: ***encapsulation***
- "Encapsulation" is the quality of variables being inaccessible outside of a particular context.
- For example, in the following code…

```
def calc (num1, num2, message):
    sum = num1 + num2
    result = message[sum]
    return result
ch = calc (1, 2, "Hello")
print (ch)
print (sum)
```

Encapsulation

- …the last line <u>**`print (sum)`**</u> would create an error because it is being interpreted in the global scope, but <u>**`sum`**</u> is only a local variable.
- Recall, the entire idea of abstraction is to make problem-solving easier by taking focus away from the smaller details.
- Encapsulation, then, is an important aspect of abstraction precisely because it <u>***hides***</u> those details
- More importantly, this protects different parts of your code from one another

Modules

- We have, in fact, already worked with modules quite a bit in this class. For example, you may recall programs with lines such as:

```
import math
import random
import time
```

- At the most basic level, a **module** is a pre-existing body of code, that can be incorporated into other code by way of *import statements*.
- Once imported, you can access the needed constants and functions through the *module name*

Modules

- In programming, there is a saying: "Do not reinvent the wheel." In other words, if a good tool already exists, don't go build a new one to do the same
- Consider the problem of calculating a square root. The algorithm is quite complex and would be a challenge to code, but the **`math`** module already has the **`sqrt`** function:

```
import math
print ("The square root of 9 is", math.sqrt(9))
print ("The square root of 16 is", math.sqrt(16))
print ("The square root of 64 is", math.sqrt(64))
```

Creating Modules

- There are many modules out there for Python. Some come with the interpreter, and some can be downloaded.
- In fact, you can write your own modules! All you need to do is:
  - Create a Python file named `module_name.py`
  - Add code for functions and constants you wish to include
  - In another code file, include the line
  - `import module_name`
- For the import to work, the module file must be accessible (for example, same folder)

Recursion

- This topic is considered to be one of the more difficult ones in introductory programming – yet it is also an essential one.
- We will not emphasize it as much in this class, but you do need to have some understanding of what it entails and when to use it.
- You should start by trying to understand the simpler examples, before tackling more complex ones
- When using recursion, you should do so mindfully.

Recursion

- To start with, think about situations in which *smaller* units are combined into *larger* – but similar – units.
- Some examples we will consider:
- A file system
- A family tree
- A discussion thread (e.g., reddit.com)
- Factorials
- The Fibonacci sequence

Recursion (Factorials)

- The *factorial* of a positive number is the product of all the integers between 1 and itself. The factorial of integer ***n*** is denoted as ***n*!** For example:

1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1

5! = 5 * 4 * 3 * 2 * 1

***n*!** = n * (n-1) * (n-2) * … * 2 * 1

Recursion (Factorials)

- Looking at this, you may notice a pattern:

1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1

5! = 5 * 4 * 3 * 2 * 1

***n*!** = n * (n-1) * (n-2) * … * 2 * 1

Recursion (Factorials)

- Therefore, you could also express these in the following manner:

1! = 1

2! = 2 * 1!

3! = 3 * 2!

4! = 4 * 3!

5! = 5 * 4!

***n*!** = n * (n-1)!

Applies to every value of n greater than one!

Recursion (Factorials)

- Part of the solution is, in fact, the solution to a smaller but similar problem.
- As such, you could write a function to compute a factorial like this:

```
def factorial (number):
    if number < 2:
        return 1
    else:
        return number * factorial (number - 1)
```

- This is a ***recursive*** function because it ***calls itself***!
- Notice that **factorial(1)** is special and simply returns 1

Recursion (Factorials)

- If you called **factorial (5)**, then the flow of control would look like this:

```
factorial (5)
  5 * factorial (4)
    4 * factorial (3)
      3 * factorial (2)
        2 * factorial (1)
              1
            2
          6
      24
  120
```

```
print (factorial (5))
>> 120
```

Recursion (Fibonacci sequence)

- The *Fibonacci sequence* is series of numbers beginning with **1** and **1**, where each subsequent number is the sum of the two previous. If we call the *n*th Fibonacci number ***f(n)***, then…

f(1) = 1

f(2) = 1

f(3) = 1 + 1 = 2

f(4) = 1 + 2 = 3

f(5) = 2 + 3 = 5

f(6) = 3 + 5 = 8

f(7) = 5 + 8 = 13

f(8) = 8 + 13 = 21

…

f(*n*) = f(n-1) + f(n-2)

(where n >= 3)

Recursion (Fibonacci sequence)

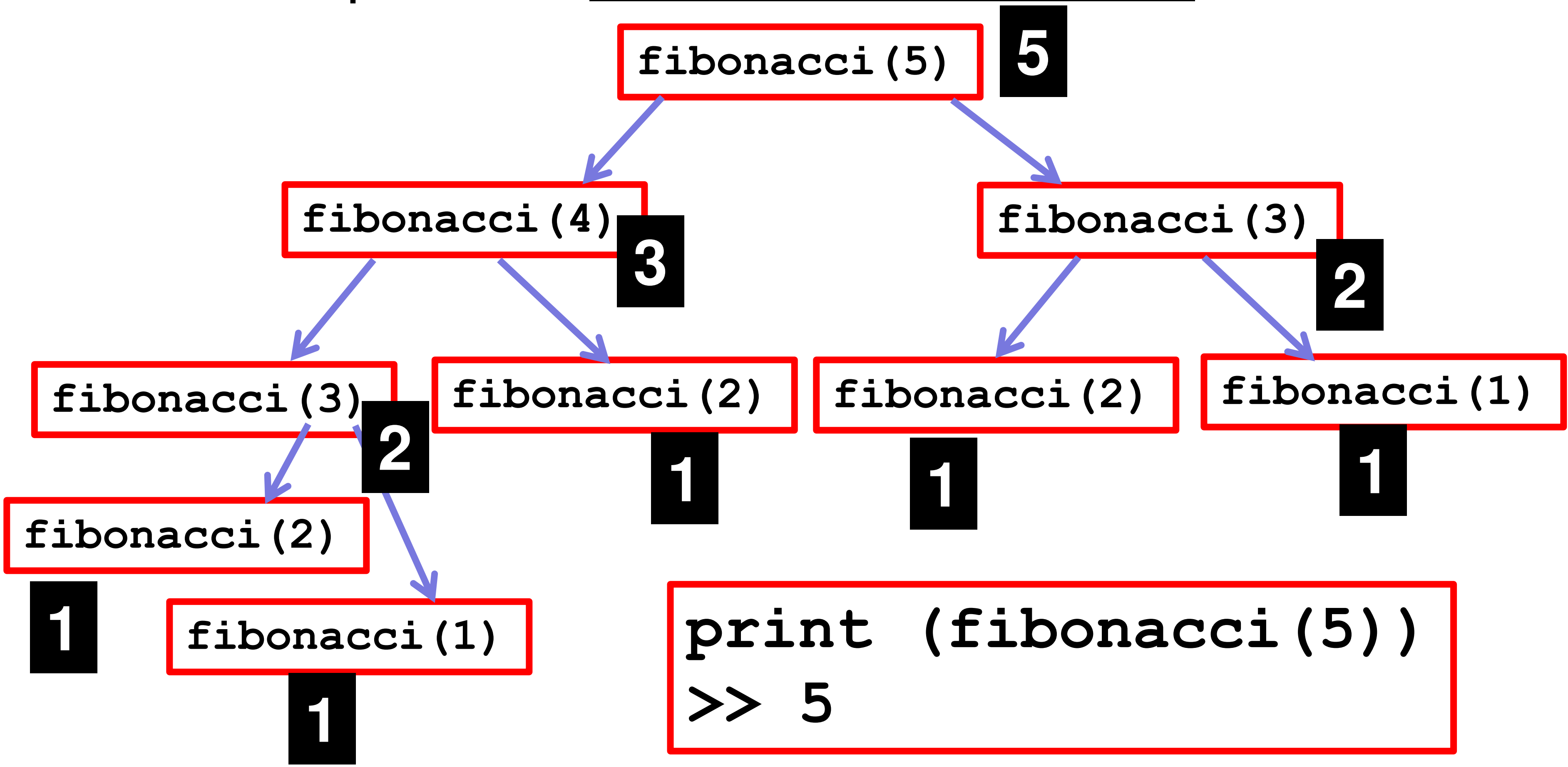
- A function to compute the ***n*th Fibonacci number** could look like this:

```
def fibonacci (num):
    if num <= 2:
        return 1
    else:
        return fibonacci(num-1) + fibonacci(num-1)
```

- Again, notice there are special cases (when num is 2 or less) where the function does not call itself but, instead, simply returns a value
- Let's look at the following example…

Recursion (Fibonacci sequence)

- Let's compute the ***5*****th** Fibonacci number:



Recursion – Concerns and Warnings

- The idea behind recursion is solving a problem by breaking it down into simpler sub-problems and combining the solutions.
- Eventually, this breaking down should stop, when you reach the simplest form of the problem. For example:
  - The factorial of 1 is simply 1
  - The first and second Fibonacci numbers are 1
- At this level, the solution is simple, requiring no further recursion.

Recursion – Concerns and Warnings

- These – **<u>1!</u>** and **<u>f(1) and f(2)</u>** – are examples of <u>*base cases*</u> in recursion
- That is, problems so small that solving them does not require calling the function again.
- A recursive function must have base cases so that the function will eventually <u>*terminate*</u>
- Otherwise, you will have <u>*infinite recursion*</u>.
- Just as you must make sure a loop eventually terminates, you must also make sure your function eventually <u>*stops calling itself*</u>

Recursion – Concerns and Warnings

- Also, when considering a recursive solution, you should ask yourself if it is the best option.
- Even if the recursion terminates, it may be undesirable *in other respects*.
- Consider **fibonacci.py**
  - It pauses for 10 milliseconds before returning
  - Lower Fibonacci numbers, such as f(5), are fast
  - However, higher ones like f(10) take **much** longer to finish calculating.
  - This is because the number of calculations, many of which are repeated, increases **exponentially!**

Recursion – Concerns and Warnings

- Here, the recursive solution consumes more...
  - Time – because more calculations must take place
  - Memory – because results are being saved in memory before finally being recombined into a final solution
- As such, this is a scenario where you would want to find a more efficient solution. Here, we will consider two such solutions:
  - Memoization
  - Iteration

Recursion Alternative – Memoization

- ***Memoization*** refers to the practice of *storing* the results of previous calculations.
- This is very applicable to the Fibonacci numbers, where many of the recursive function calls repeat previous calculations. Consider **fibonacci_dict.py**
  - Here, we create an empty dictionary
  - Every time we calculate the ***n*****th** Fibonacci number, we add an entry to the dictionary, with ***n*** as the key
  - Thus, if the dictionary already contains the ***n*****th** Fibonacci number, then we simply fetch it
  - **f(15)**, for example, will not require us to recalculate **f(13)** and **f(14)**

Recursion Alternative – Iteration

- ***Iteration*** simply means repetition or *looping*.
- Calculating **f(n)** simply requires the values of **f(n-1)** and **f(n-2)**, so we could just use three variables and a loop. Consider **fibonacci_loop.py**
  - We have variables for three different values: the result, **f(n-1),** and **f(n-2)**
  - Every time we calculate the ***n*th** Fibonacci number, overwrite the values for the two previous and store the current in the **result** variable.
  - Like the dictionary version, calculating one value *does not* require the recalculation of previous results, so it is much quicker!

Recursion – When to Use It

- That said, there will be several scenarios where a **recursive** option is superior.
  - Recursion reduces the problem size.
    - ➢ Searching a sorted sequence
  - The recursive option is more intuitive (without being inefficient in implementation)
    - ➢ Exploring a tree structure
  - The recursive option is more efficient
    - ➢ Sorting an unsorted sequence
- It's a case where you'll have to make a decision…

Other Topics in Functions

- **Docstrings:**
  - ➢ As the first line in your function, you can include a triple-quoted comment about the function.
  - ➢ It will not directly affect the function's behavior, but…
  - ➢ …it can be helpful to you and other coders.
  - ➢ Some IDEs, such as IDLE, may make use of it.
- **Positions of parameters:**
  - Normally, when calling a function, you must provide the right number of values in the right order
  - At runtime, Python will attempt to interpret the call

Other Topics in Functions

- **Positions of parameters:**
  - Supplying parameters incorrectly can create runtime and/or logic errors
  - However, there are other options, as well
- **Keyword arguments:**
  - If you know the *formal* parameters' names, then you can supply keyword arguments. Consider this method:

```
def full_name(first, last):
   return first + " " + last
```

  - To start with, let's look at standard behavior…

Other Topics in Functions

- **Keyword arguments:**

```
def full_name(first, last):
    return first + " " + last
```

```
print (full_name ("John", "Doe"))
```

```
John Doe
```

```
print (full_name ("Doe", "John"))
```

```
Doe John
```

- However, this…

```
print (full_name (last="Doe", first="John"))
```

will print as…

```
John Doe
```

Other Topics in Functions

- **Keyword arguments:**

```
def full_name(first, last):
    return first + " " + last
```

```
print (full_name (last="Doe", first="John"))
```

  - Here, the keywords *override* the order of the parameters
  - If you use keywords for one parameter, then you must use them for all parameters!

- **Default parameter values:**
  - When writing a function, you may find it helpful to assign *default values* for the parameters
  - This can make the function simpler to use…

Other Topics in Functions

- **Default parameter values:**
  - This can make the function simpler to use…
  - …while also providing some degree of flexibility
  - Consider this variation on the **full_name** function:

```
def full_name(first="John", last="Doe"):
    return first + " " + last
```

  - You can use the parameters in many ways and have the function behave differently
  - Call with no values; use all default values:

```
>>> print (full_name())
John Doe
```

Other Topics in Functions

- **Default parameter values:**

```
def full_name(first="John", last="Doe"):
    return first + " " + last
```

  - Supply both parameters without keywords (will impose order of formal parameter list):

```
>>> print (full_name("Jane", "Smith"))
Jane Smith
>>> print (full_name("Smith", "Jane"))
Smith Jane
```

  - Supply both parameters **with** keywords:

```
>>> print (full_name(first="Jane", last="Smith"))
Jane Smith
```

Other Topics in Functions

- **Default parameter values:**

```
def full_name(first="John", last="Doe"):
    return first + " " + last
```

  - Supply both parameters **with** keywords (keywords override the order):

```
>>> print (full_name(last="Smith", first="Jane"))
Jane Smith
```

  - **Supply** one parameter and **allow default** for other:

```
>>> print (full_name(first="Jane"))
Jane Doe
>>> print (full_name(last="Smith"))
John Smith
```