

# PYTHON PROGRAMMING

2018

# Contents

---

1. Functions
2. Recursion

# Functions

## □ Objectives

- ▣ To define functions with formal parameters.
- ▣ To invoke functions with actual parameters (i.e., arguments).
- ▣ To distinguish between functions that return and do not return a value.
- ▣ To invoke a function using positional arguments or keyword arguments.
- ▣ To pass arguments by passing their reference values.
- ▣ To develop reusable code that is modular and is easy to read, debug, and maintain.
- ▣ To create modules for reusing functions.
- ▣ To determine the scope of variables.
- ▣ To define functions with default arguments.
- ▣ To define a function that returns multiple values.
- ▣ To apply the concept of function abstraction in software development.
- ▣ To design and implement functions using stepwise refinement.
- ▣ To simplify drawing programs with reusable functions.

# 1 Introduction

---

- Functions can be used to define reusable code and organize and simplify code.

## 2 Defining a Function

- A function is a collection of statements grouped together that performs an operation.
- A function definition consists of the function's name, parameters and body.
- The syntax for defining a function

```
def functionName(list of parameters)  
    # Function body
```

# 2 Defining a Function

## □ Example

- ▣ A function created to find which of two numbers is bigger.

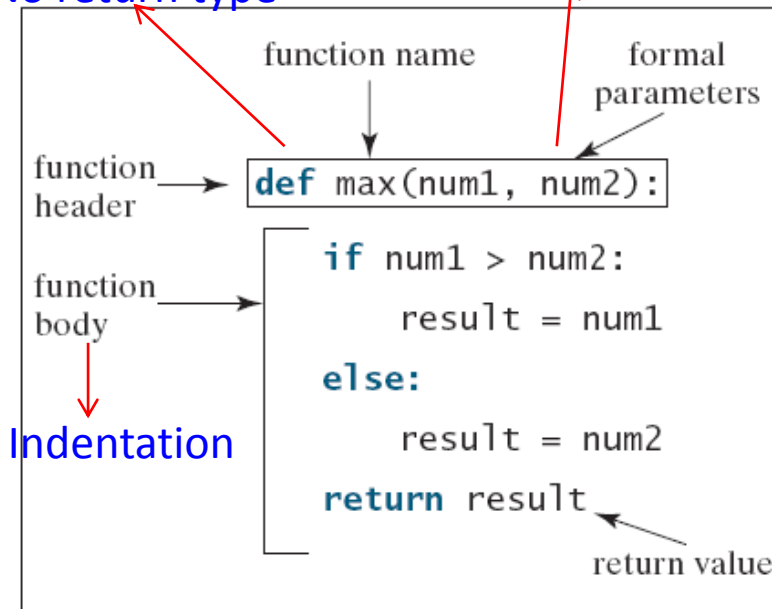
Return type is determined by the return statement

Data type of parameter is determined by the arguments of the calling function

Define a function

No return type

No data type



Invoke a function

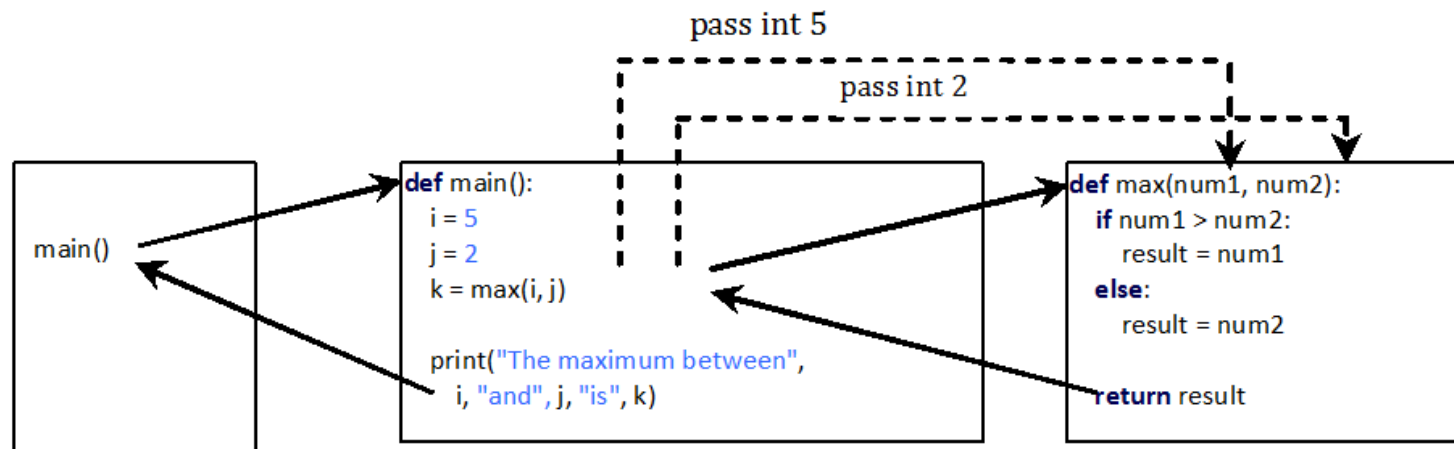
```
z = max(x, y)
```

↑ ↑  
actual parameters  
(arguments)

- The function header begins with the **def** keyword, followed by function's name and parameters, ends with a **colon**.

# 3 Calling a Function

## □ Example(TestMax)

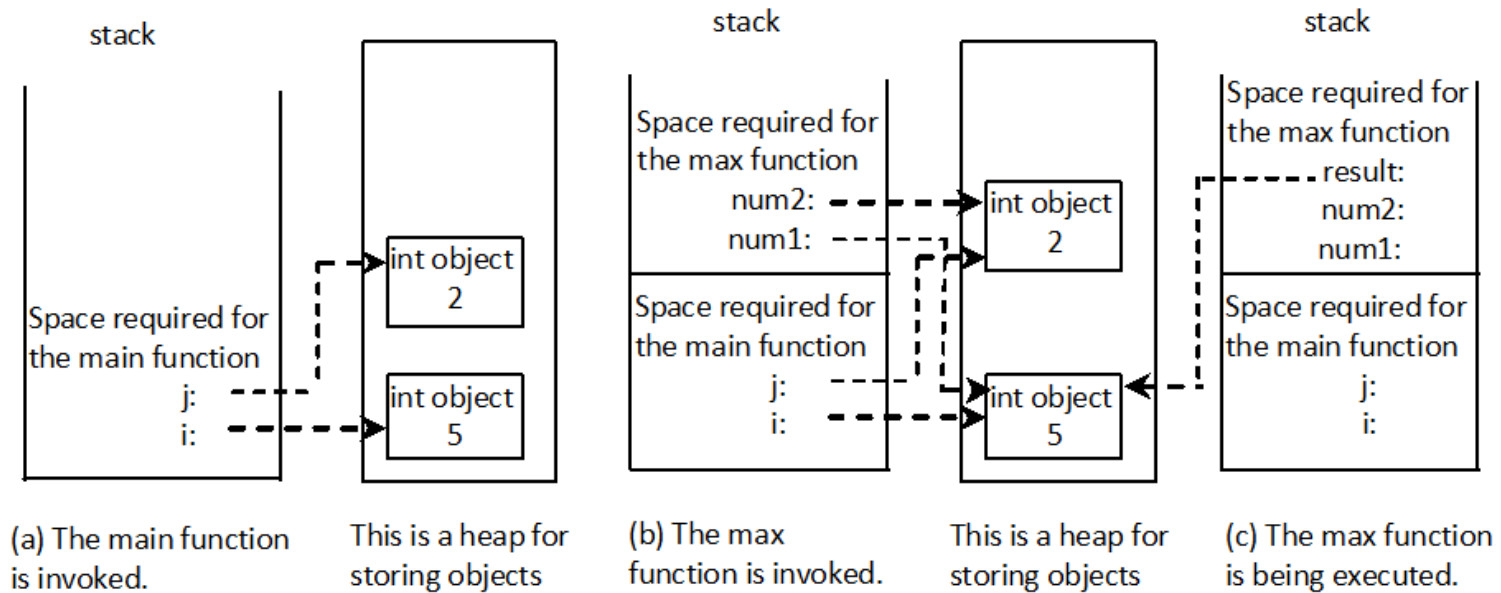


# 3.1 Call Stacks

## □ Call stack

- ▣ Each time a function is invoked, the system creates an **activation record** that stores its arguments and variables for the function and places the **activation record** in a call stack.
- ▣ When a function calls another function, the caller's activation record is kept intact and a new activation record is created for the new function call.
- ▣ When a function finishes its work and returns control to its caller, its activation record is removed from the call stack.
- ▣ A call stack stores the activation records in a **last-in, first-out** fashion.





# 4 Functions with/without Return Values

- void function
  - ▣ A function does not have to return a value.
  - ▣ Example(PrintGradeFunction)
- Example(ReturnGradeFunction)

# 4 Functions with/without Return Values

## □ The None Value

- ▣ Technically, every function in Python returns a value whether you use return or not.
- ▣ If a function does not return a value, by default, it returns a special value **None**.

# 4 Functions with/without Return Values

## □ The None Value

### ▣ Example

```
def sum(number1, number2):  
    total = number1 + number2  
print(sum(1, 2))
```

### ▣ Output

None

# 5 Positional and Keyword Arguments

- A function's arguments can be passed as positional arguments or keyword arguments.

# 5 Positional and Keyword Arguments

## □ Passing Arguments by Positions

- ▣ The arguments should be passed in the same order as their respective parameters in the function header.
- ▣ The arguments must match the parameters in **order**, **number**, and **compatible type**, as defined in the function header.

### ▣ Example

```
def nPrintln(message, n):  
    for i in range(0, n):  
        print(message)
```

Invoke by



```
nPrintln("Welcome to Python", 5)
```

```
nPrintln("Computer Science", 15)
```

```
nPrintln(4, "Computer Science")
```

# 5 Positional and Keyword Arguments

## □ Keyword Arguments

▣ Pass each argument in the form `name = value`

▣ Example

```
def nPrintln(message, n):  
    for i in range(0, n):  
        print(message)
```



What is wrong?

```
nPrintln(4, "Computer Science")
```

Is this OK?

```
nPrintln(n = 4, message = "Computer Science")
```

# 6 Passing Arguments by Reference Values

## □ Pass by Value

- ▣ In Python, all data are objects.
- ▣ A variable for an object is actually a reference to the object.
- ▣ When invoke a function with a parameter, the reference value of the argument is passed to the parameter.
- ▣ If the argument is a number or a string, the argument is not affected, regardless of the changes made to the parameter inside the function.



# 6 Passing Arguments by Reference Values

## □ Pass by Value

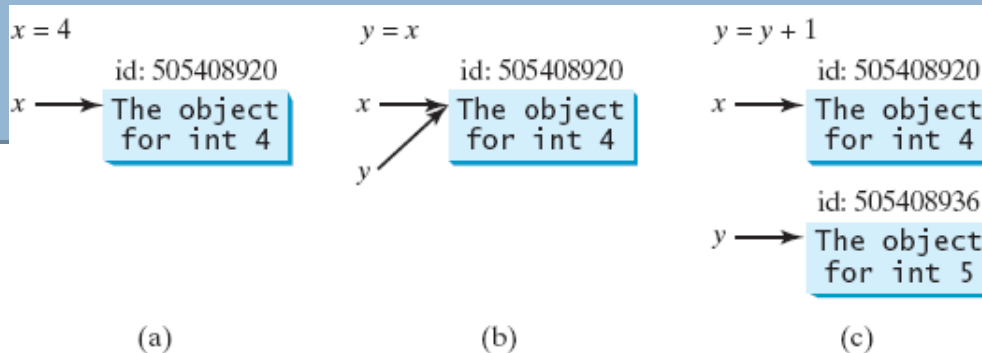
### ▣ Example(Increment)

- The reason is that numbers and strings are known as **immutable objects**.
- The contents of immutable objects cannot be changed.
- Whenever assign a new number to a variable, Python creates a new object for the new number and assigns the reference of the new object to the variable.

# 6 Passing Arguments by Reference Values

```
>>> x = 4
>>> y = x
>>> id(x)           # The reference of x
505408920
>>> id(y)           # The reference of y is the same as the reference of x
505408920
>>>
```

```
>>> y = y + 1 # y now points to a new int object with value 5
>>> id(y)
505408936
>>>
```



(a) 4 is assigned to x; (b) x is assigned to y; (c) y + 1 is assigned to y.

# 7 Modularizing Code

- Modularizing Code
  - ▣ Functions can be used to reduce redundant coding and enable code reuse.
  - ▣ Functions can also be used to modularize code and improve the quality of the program.

# 7 Modularizing Code

## □ Modularizing Code

- ▣ In Python, the function definition can be placed into a file called **module** with the file-name extension **.py**.
- ▣ The module can be later imported into a program for reuse.
- ▣ The module file should be placed in the same directory with the other programs.
- ▣ A module can contain more than one function.
- ▣ Each function in a module must have a different name.

# 7 Modularizing Code

## □ Example(GCDFFunction TestGCDFFunction)

### ▣ Note

- `import GCDFFunction` can also be used.
- But `GCDFFunction.gcd` should be used to invoke the `gcd` function.

# 7 Modularizing Code

- Several advantages by encapsulating the code for obtaining the `gcd` in a function
  - ▣ It isolates the problem for computing the `gcd` from the rest of the code in the program.
    - Thus, the logic becomes clear and the program is easier to read.
  - ▣ Any errors for computing the `gcd` are confined to the `gcd` function, which narrows the scope of debugging.
  - ▣ The `gcd` function now can be reused by other programs.

# 7 Modularizing Code

## □ Example(PrimeNumberFunction)

- ▣ This program divides a large problem into two subproblems.
  - As a result, the new program is easier to read and easier to debug.
- ▣ Moreover, the functions `printPrimeNumbers` and `isPrime` can be reused by other programs.

# 8 Case Study

---

- Converting Decimals to Hexadecimals  
(Decimal2HexConversion)



# 9 The Scope of Variables


## □ Scope

- ▣ The part of the program where the variable can be referenced.
- ▣ Local variable
  - A variable created inside a function
  - Can only be accessed inside a function
  - The scope starts from its creation and continues to the end of the function that contains the variable.
- ▣ Global variable
  - Created outside all functions and are accessible to all functions in their scope.

# 9 The Scope of Variables

```
globalVar = 1
def f1():
    localVar = 2
    print(globalVar)
    print(localVar)
f1()
print(globalVar)
print(localVar) # Out of scope. This gives an error
```

```
x = 1
def f1():
    x = 2
    print(x) # Displays 2
f1()
print(x) # Displays 1
```



The global variable **x** is not accessible in the function. Outside the function, the global variable **x** is still accessible.

# 9 The Scope of Variables

```
x = eval(input("Enter a number: "))  
if (x > 0):  
    y = 4  
print(y) # This gives an error if y is not created
```

# 10 Default Arguments

## □ Default Arguments

- ▣ Python allows to define functions with default argument values.
- ▣ The default values are passed to the parameters when a function is invoked without the arguments.
- ▣ `functionName._defaults_` can be used to display all the current value of the default arguments of **functionName**.
- ▣ Example(**DefaultArgumentDemo**)

### ■ Note

- A function may mix parameters with default arguments and non-default arguments.
- In this case, the non-default parameters must be defined before default parameters.

# 10 Default Arguments

## □ Default Arguments

- ▣ The default values are evaluated at the point of function definition in the defining scope

■ Output **5**

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

# 10 Default Arguments

## □ Important warning

- ▣ The default value is evaluated only once.

- This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

# 10 Default Arguments

## □ Example

- ▣ The following function accumulates the arguments passed to it on subsequent calls

■ Output

```
[1]  
[1, 2]  
[1, 2, 3]
```

- ▣ If we don't want the default to be shared between subsequent calls, we can write the function like

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

```
def f(a, L=[]):  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```

# 11 Returning Multiple Values

- Returning Multiple Values
  - ▣ The Python **return** statement can return multiple values.
  - ▣ Example(MultipleReturnValueDemo)
    - The **sort** function returns two values.
    - When it is invoked, the returned values need to be passed in a simultaneous assignment.

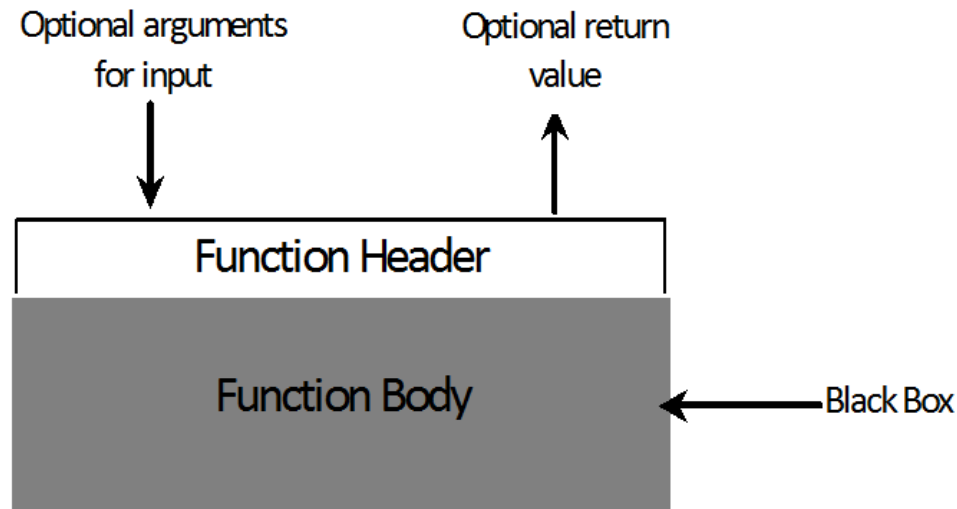


# 12 Case Study

- Generating Random ASCII Characters  
(RandomCharacter) (TestRandomCharacter)
  - ▣ A character is coded using an integer.
  - ▣ Generating a random character is to generate an integer.

# 13 Function Abstraction and Stepwise Refinement

- Function Abstraction
  - ▣ It separates the use of a function from its implementation.
  - ▣ Known as **information hiding** or **encapsulation**.
  - ▣ The function body can be thought as a black box that contains the detailed implementation for the function.



# 13 Function Abstraction and Stepwise Refinement

- Benefits of Functions
  - ▣ Write a function once and reuse it anywhere
  - ▣ Information hiding
    - Hide the implementation from the user
  - ▣ Reduce complexity

# 13 Function Abstraction and Stepwise Refinement

- Stepwise Refinement
  - ▣ Also known as “**divide and conquer**” strategy
  - ▣ When writing a large program, to decompose it into subproblems.
    - And the subproblems can be further decomposed into smaller, more manageable problems.

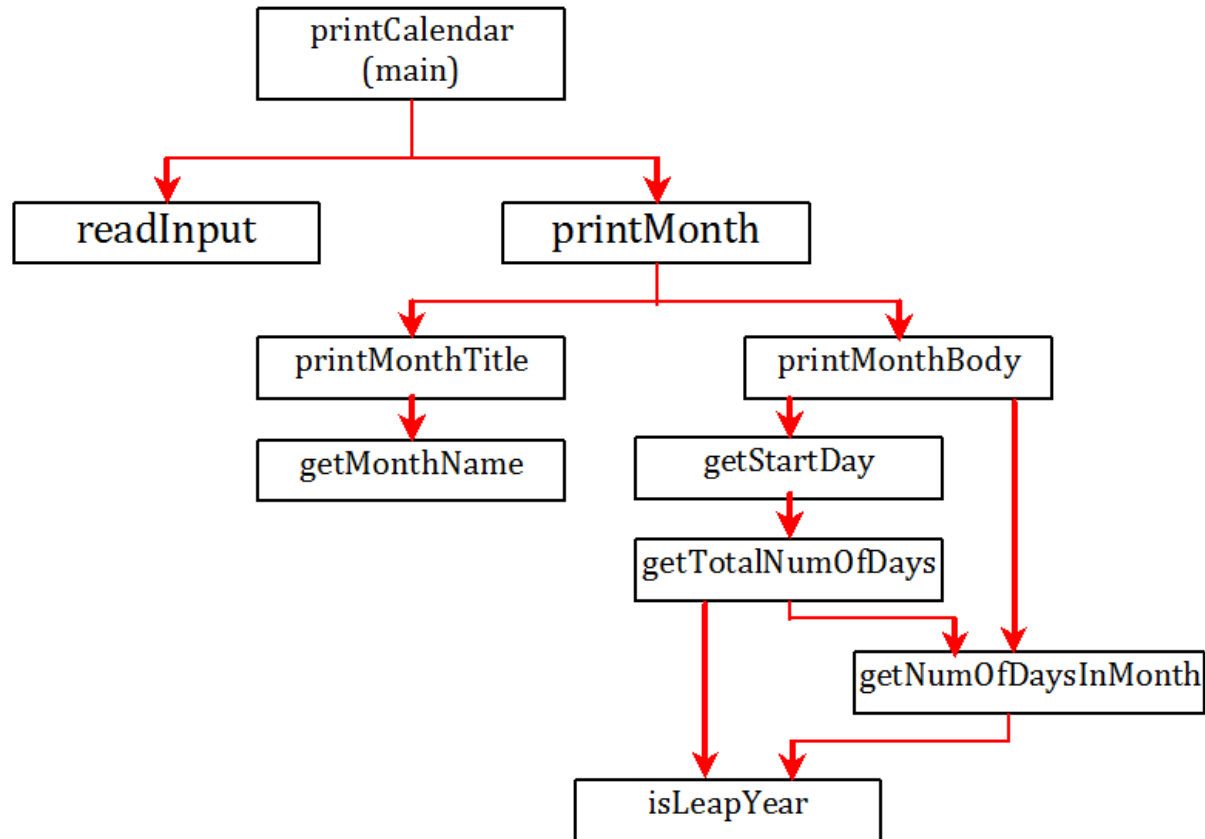
# 13 Function Abstraction and Stepwise Refinement

- Case study (`PrintCalendar`)
  - ▣ Write a program that displays the calendar for a given month of the year.

```
Enter full year (e.g., 2001): 2011 ↵Enter
Enter month as number between 1 and 12: 9 ↵Enter

September 2011
-----
Sun  Mon  Tue  Wed  Thu  Fri  Sat
    4   5   6   7   1   2   3
11  12  13  14  15  16  17
18  19  20  21  22  23  24
25  26  27  28  29  30
```

# 13.1 Top-Down Design



# 13.2 Top-Down and/or Bottom-Up Implementation

- Top-Down Implementation
  - ▣ It implements one function in the structure chart at a time from the top to the bottom.
  - ▣ **Stubs** can be used for the functions waiting to be implemented.
    - A stub is a simple but incomplete version of a function.
    - The use of stubs enables to test invoking the function from a caller.
  - ▣ Implement the main function first and then use a stub for the printMonth function.

```

# A stub for printMonth may look like this
def printMonth(year, month):
    print(year, month)

# A stub for printMonthTitle may look like this
def printMonthTitle(year, month):
    print("printMonthTitle")

# A stub for getMonthBody may look like this
def getMonthBody(year, month):
    print("getMonthBody")

# A stub for getMonthName may look like this
def getMonthName(month):
    print("getMonthName")

# A stub for getStartDay may look like this
def getStartDay(year, month):
    print("getStartDay")

# A stub for getTotalNumberOfDays may look like this
def getTotalNumberOfDays(year, month):
    print("getTotalNumberOfDays")

# A stub for getNumberOfDaysInMonth may look like this
def getNumberOfDaysInMonth(year, month):
    print("getNumberOfDaysInMonth")

# A stub for isLeapYear may look like this
def isLeapYear(year):
    print("isLeapYear")

def main():
    # Prompt the user to enter year and month
    year = eval(input("Enter full year (e.g., 2001): "))
    month = eval(input("Enter month as number between 1 and 12: "))
    # Print calendar for the month of the year
    printMonth(year, month)

```



# 13.2 Top-Down and/or Bottom-Up Implementation

- Bottom-Up Implementation
  - ▣ It implements one function in the structure chart at a time from the bottom to the top.
  - ▣ For each function implemented, write a test program to test it.

# 13.4 Benefits of Stepwise Refinement

---

- Simpler Program
- Reusing Functions
- Easier Developing, Debugging, and Testing
- Better Facilitating Teamwork

# 14 Case Study

- Reusable Graphics Functions ([UsefulTurtleFunctions](#))
  - ▣ Often we need to
    - Draw a line between two points
    - Display text or a small point at a specified location
    - Depict a circle with a specified center and radius
    - Create a rectangle with a specified center, width, and height
  - ▣ It would greatly simplify programming if these functions were available for reuse.
  - ▣ Use the above functions to draw shapes ([UseCustomTurtleFunctions](#))

# 14 Case Study

- Recur
- Reusable Graphics Functions ([UsefulTurtleFunctions](#))
  - ▣ Often we need to
    - Draw a line between two points
    - Display text or a small point at a specified location
    - Depict a circle with a specified center and radius
    - Create a rectangle with a specified center, width, and height
  - ▣ It would greatly simplify programming if these functions were available for reuse.
  - ▣ Use the above functions to draw shapes ([UseCustomTurtleFunctions](#))

# Recursion

## □ Objectives

- ▣ To describe what a recursive function is and the benefits of using recursion.
- ▣ To develop recursive functions for recursive mathematical functions.
- ▣ To explain how recursive function calls are handled in a call stack.
- ▣ To use a helper function to derive a recursive function.
- ▣ To solve selection sort using recursion.
- ▣ To solve binary search using recursion.
- ▣ To get the directory size using recursion.
- ▣ To solve the Towers of Hanoi problem using recursion.
- ▣ To draw fractals using recursion.
- ▣ To solve the Eight Queens problem using recursion.
- ▣ To discover the relationship and difference between recursion and iteration.
- ▣ To know tail-recursive functions and why they are desirable.

# 1 Introduction

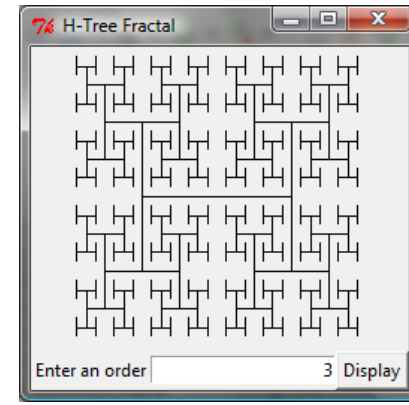
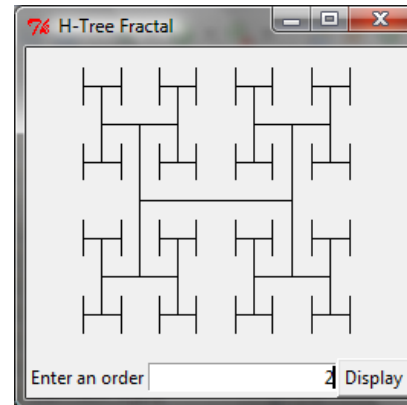
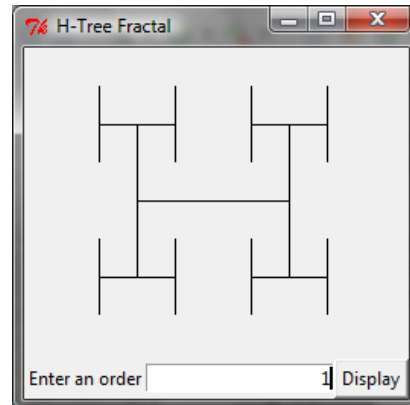
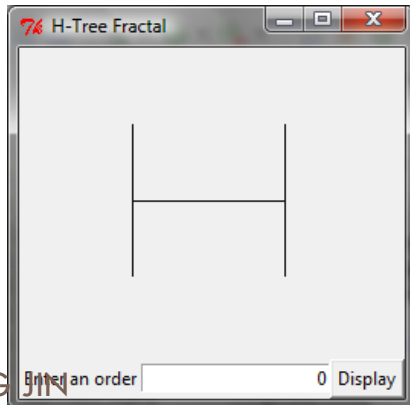
## □ Motivation

- ▣ Suppose we want to find all the files under a directory that contains a particular word.
- ▣ How do we solve this problem?
- ▣ An intuitive solution is to use recursion by searching the files in the subdirectories recursively.

# 1 Introduction

## □ Motivation

- The H-tree is used in a very large-scale integration (VLSI) design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays.
- How do we write a program to display the H-tree?
- A good approach is to use recursion by exploring the recursive pattern.



# 1 Introduction

## □ Recursion

- ▣ A technique that leads to elegant solutions to problems that are difficult to program using simple loops.



## 2 Case Study

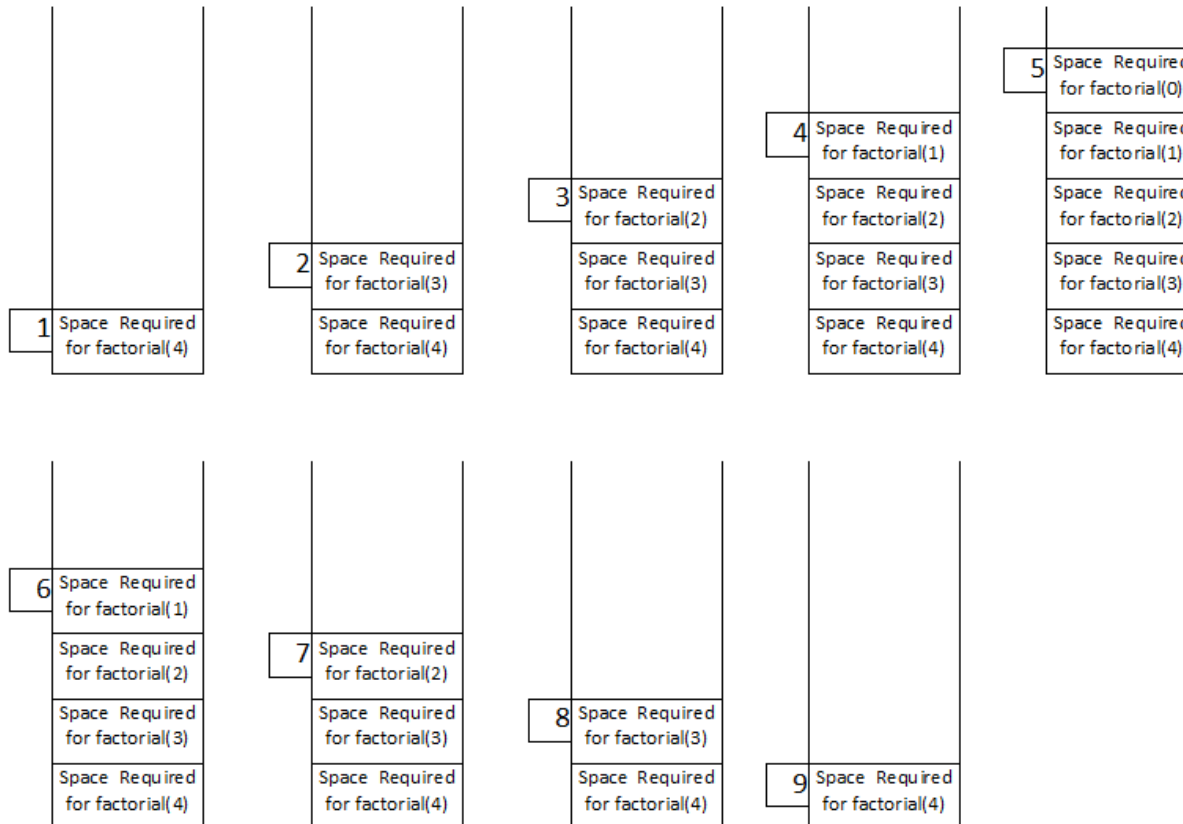
### □ Computing Factorials(`ComputeFactorial`)

- ▣ The factorial of a number  $n$  can be recursively defined as

$$\begin{aligned} 0! &= 1; \\ n! &= n \times (n - 1)!; n > 0 \end{aligned}$$

# 2 Case Study

## factorial(4) Stack Trace



# 3 Case Study

## □ Computing Fibonacci Numbers(ComputeFibonacci)

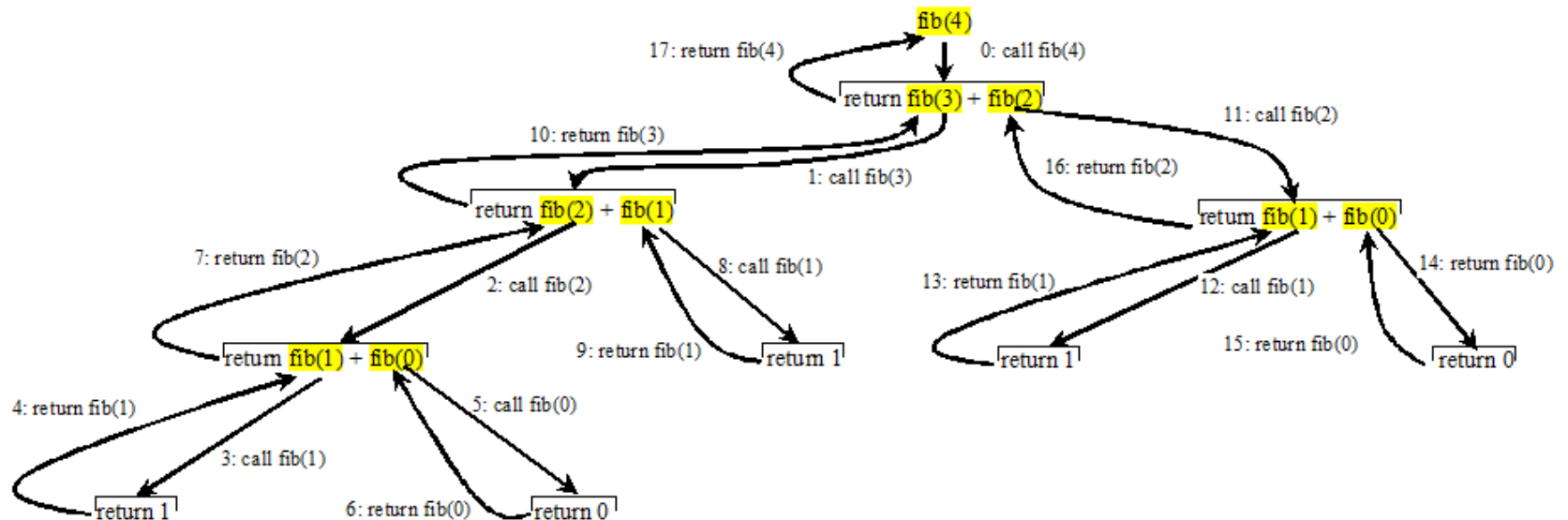
### ▣ The well-known Fibonacci-series problem

The series:	0	1	1	2	3	5	8	13	21	34	55	89	...
indexes:	0	1	2	3	4	5	6	7	8	9	10	11	

```
fib(0) = 0;  
fib(1) = 1;  
fib(index) = fib(index -1) + fib(index -2); index >=2
```

# 3 Case Study

## □ Computing Fibonacci Numbers(ComputeFibonacci)



# 4 Problem Solving Using Recursion

## □ Characteristics of Recursion

- ▣ The function is implemented using an **if-else** or a **switch** statement that leads to different cases.
- ▣ One or more base cases (the simplest case) are used to stop recursion.
- ▣ Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

# 4 Problem Solving Using Recursion

## □ Think Recursively

### ▣ Palindrome problem([RecursivePalindromeUsingSubstring](#))

- A string is a palindrome if it reads the same from the left and from the right.

1. Determine whether the first character and the last character of the string are equal.
2. Ignore the two end characters and see if the rest of the substring is a palindrome.

# 5 Recursive Helper Functions

## □ Problem

- ▣ The preceding recursive isPalindrome method is not efficient, because it creates a new string for every recursive call.
- ▣ To avoid creating new strings, use a helper method (RecursivePalindrome)

# 5 Recursive Helper Functions

## 1. Recursive Selection Sort (`RecursiveSelectionSort`)

- ① Find the smallest element in the list and swap it with the first element.
- ② Ignore the first element and sort the remaining smaller list recursively.



# 5 Recursive Helper Functions

## 2. Recursive Binary Search (`RecursiveBinarySearch`)

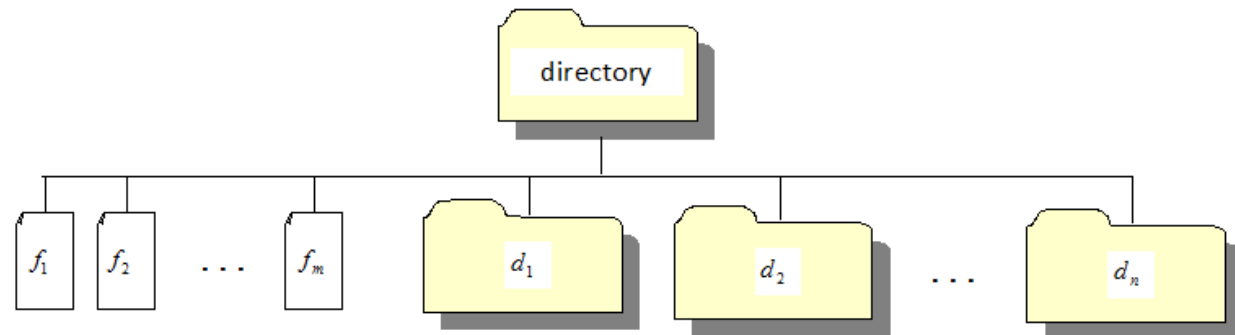
- Case 1: If the key is less than the middle element, the program recursively searches for the key in the first half of the list.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, the program recursively searches for the key in the second half of the list.

# 6 Case Study

## □ Finding the Directory Size (DirectorySize)

### ▣ The problem is

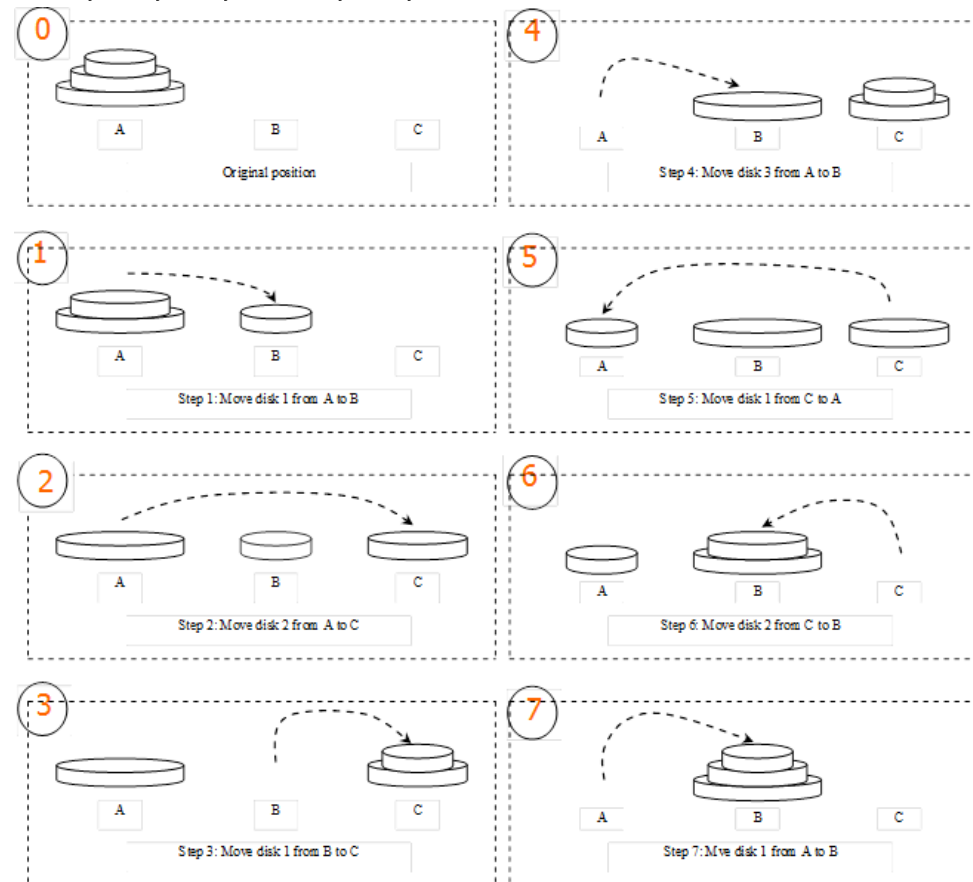
- To find the size of a directory.
- The size of a directory is the sum of the sizes of all the files in the directory.
- A directory **d** may contain subdirectories.
- Suppose a directory contains files  $f_1, f_2, \dots, f_m$  and subdirectories  $d_1, d_2, \dots, d_n$ .



# 7 Case Study

## □ Towers of Hanoi (TowersOfHanoi)

- ▣ There are  $n$  disks labeled  $1, 2, 3, \dots, n$ , and three towers labeled A, B, and C.
- ▣ No disk can be on top of a smaller disk at any time.
- ▣ All the disks are initially placed on tower A.
- ▣ Only one disk can be moved at a time, and it must be the top disk on

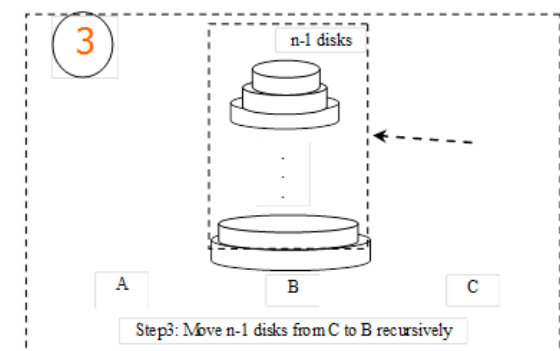
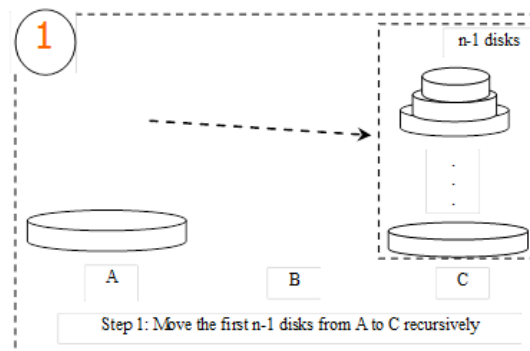
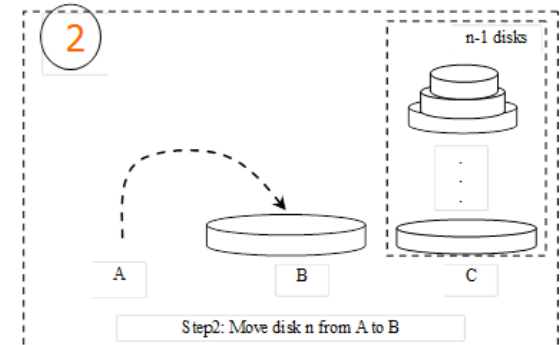
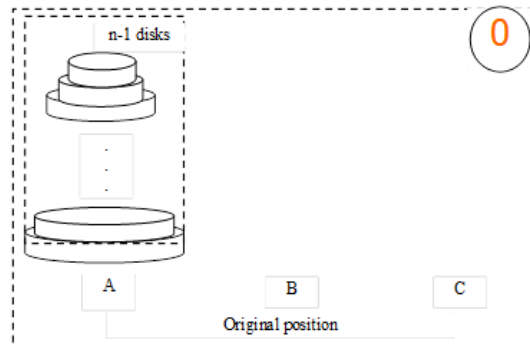


# 7 Case Study

## □ Towers of Hanoi (TowersOfHanoi)

### ▣ Solution to Towers of Hanoi

- Move the first  $n - 1$  disks from A to C with the assistance of tower B.
- Move disk  $n$  from A to B.
- Move  $n - 1$  disks from C to B with the assistance of tower A.

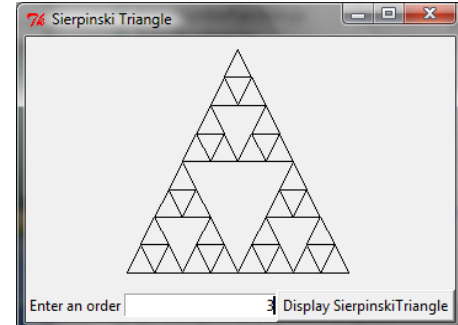
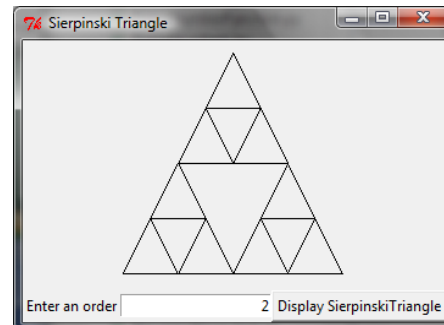
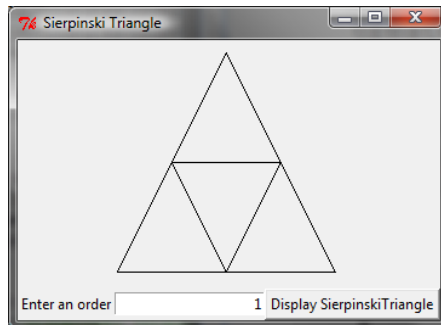
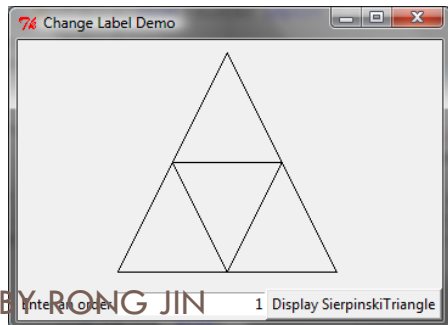


# 8 Case Study

## □ Fractals (SierpinskiTriangle)

- ▣ A fractal is a geometrical figure just like triangles, circles, and rectangles.
- ▣ But fractals can be divided into parts, each of which is a reduced-size copy of the whole.
- ▣ There are many interesting examples of fractals.
  - Sierpinski triangle

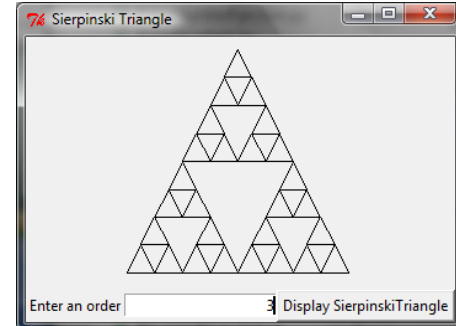
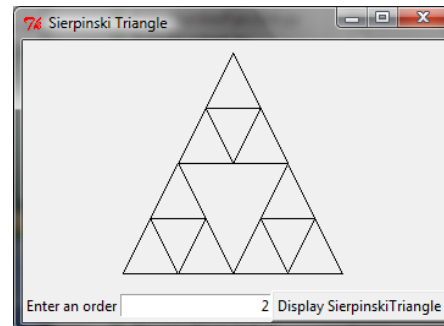
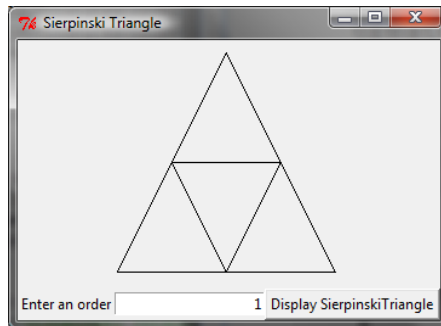
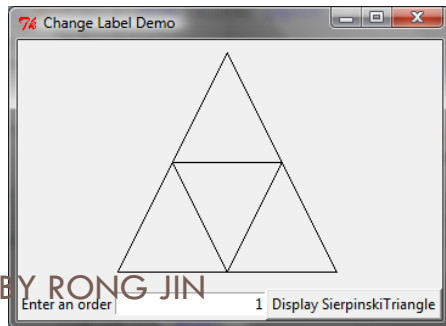
A simple fractal, named after a famous Polish mathematician.



# 8 Case Study

## □ Fractals (SierpinskiTriangle)

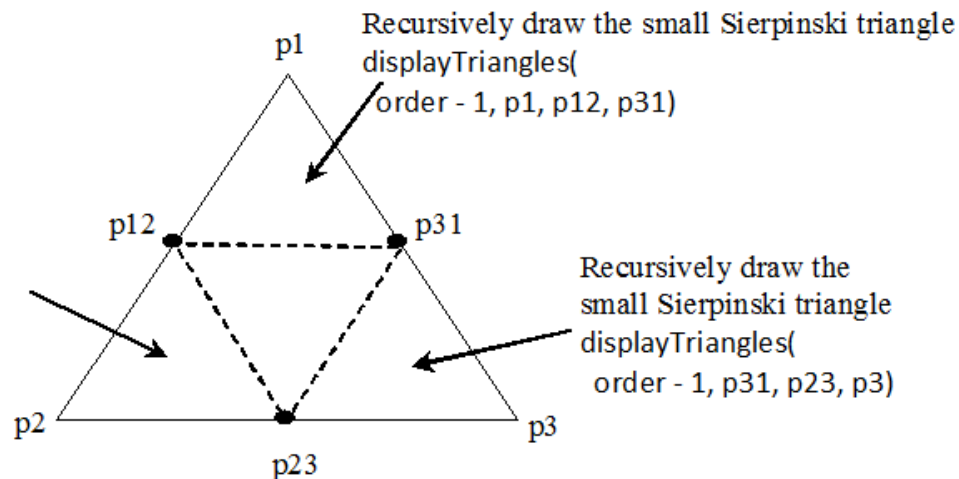
- It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0.
- Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1.
- Leave the center triangle intact.  
Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2.
- Repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on.



# 8 Case Study

## □ Fractals (SierpinskiTriangle)

Recursively draw the small  
Sierpinski triangle  
`displayTriangles(  
order - 1, p12, p2, p23)`



# 8 Case Study

- Koch curve (DrawKoch)(DrawKoch2)
  - ▣ A mathematical curve and one of the earliest fractal curves to have been described.

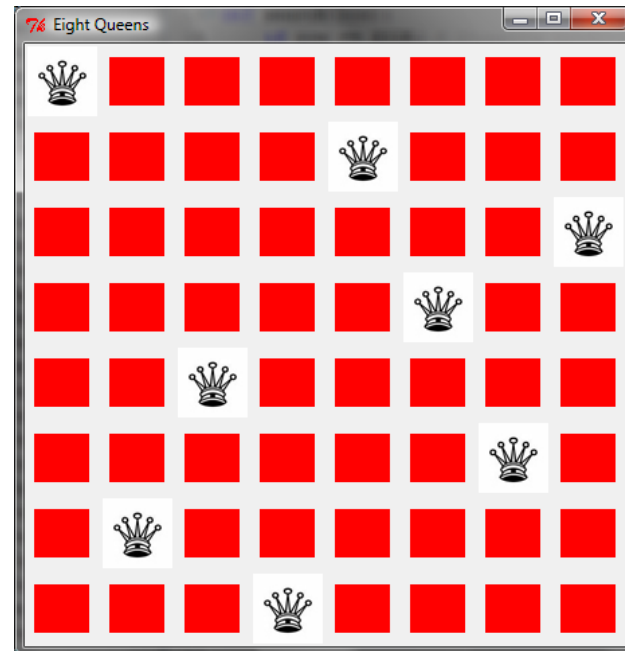


# 9 Case Study

## □ Eight Queens (EightQueens)

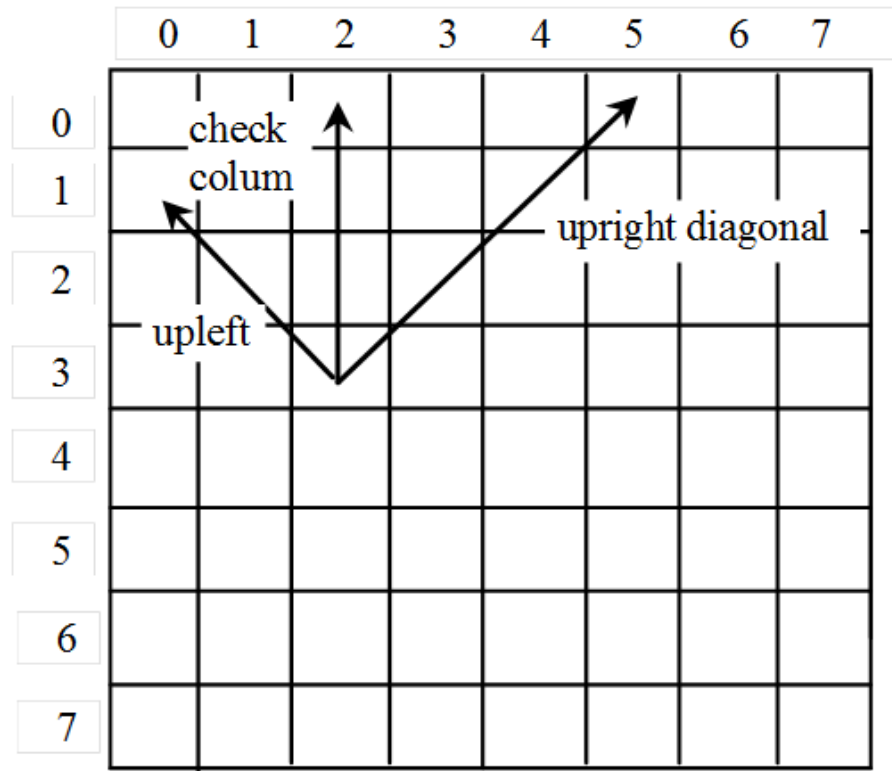
- ▣ To place a queen in each row on a chessboard so that no two queens can attack each other.
- There can only be one queen in each row, and the queens must be positioned such that no two queens can take the other.

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3



# 9 Case Study

## □ Eight Queens (EightQueens)



# 10 Recursion vs. Iteration

## □ Recursion

- ▣ An alternative form of program control.
- ▣ Essentially repetition without a loop.
- ▣ Recursion bears substantial overhead.
- ▣ Each time the program calls a method, the system must assign space for all of the method's local variables and parameters.
- ▣ This can consume considerable memory and requires extra time to manage the additional space.

# 10 Recursion vs. Iteration

## □ Iteration

- ▣ Any problem that can be solved recursively can be solved nonrecursively with iterations.
- ▣ Recursion is good for solving the problems that are inherently recursive.

# Lambda Expressions

## □ Lambda Expressions

- ▣ Small anonymous functions can be created with the **lambda** keyword.
- ▣ This function returns the sum of its two arguments:  
**lambda a, b: a+b**
- ▣ Lambda functions can be used wherever function objects are required.
- ▣ They are syntactically restricted to a single expression.
- ▣ Semantically, they are just syntactic sugar for a normal function definition.

# Lambda Expressions

## □ Lambda Expressions

- Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):  
...     return lambda x: x + n  
...  
>>> f = make_incrementor(42)  
>>> f(0)  
42  
>>> f(1)  
43
```

- The above example uses a lambda expression to return a function.

# Lambda Expressions

## □ Lambda Expressions

- ▣ Another use is to pass a small function as an argument:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

# Datetime Module

---

1. Available Types
2. datetime Objects



# Datetime Module

- The datetime module
  - ▣ It supplies classes for manipulating dates and times in both simple and complex ways.
  - ▣ It exports the following constants
    - **datetime.MINYEAR**
      - The smallest year number allowed in a date or datetime object.
      - MINYEAR is 1.
    - **datetime.MAXYEAR**
      - The largest year number allowed in a date or datetime object.
      - MAXYEAR is 9999.

# 1 Available Types

- class **datetime.date**

- ▣ An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect.
- ▣ Attributes: year, month, and day.

- class **datetime.time**

- ▣ An idealized time, independent of any particular day, assuming that every day has exactly  $24 \times 60 \times 60$  seconds (there is no notion of “leap seconds” here).
- ▣ Attributes: hour, minute, second, microsecond, and tzinfo.

# 1 Available Types

- class **datetime.datetime**

- ▣ A combination of a date and a time.
- ▣ Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

- class **datetime.timedelta**

- ▣ A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

# 1 Available Types

- class **datetime.tzinfo**

- ▣ An abstract base class for time zone information objects.
- ▣ These are used by the **datetime** and **time** classes to provide a customizable notion of time adjustment.
  - For example, to account for time zone and/or daylight saving time.

- class **datetime.timezone**

- ▣ A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

## 2 datetime Objects

- Datetime object
  - ▣ A single object containing all the information from a date object and a time object.
  - ▣ Datetime assumes there are exactly  $3600 \times 24$  seconds in every day.

# 2 datetime Objects

## □ Constructor

```
class datetime.datetime(year, month, day, hour=0, minute=0, second=0,  
                        microsecond=0, tzinfo=None, *, fold=0)
```

- The **year**, **month** and **day** arguments are required.
- **tzinfo** may be None, or an instance of a tzinfo subclass.
- The remaining arguments may be integers, in the following ranges

$\text{MINYEAR} \leq \text{year} \leq \text{MAXYEAR}$

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq \text{number of days in the given month and year}$

$0 \leq \text{hour} < 24$

$0 \leq \text{minute} < 60$

$0 \leq \text{second} < 60$

$0 \leq \text{microsecond} < 1000000$

fold in  $[0, 1]$

# 2 datetime Objects

## □ Class methods

### ▣ `datetime.today()`

- Return the current local datetime, with tzinfo None.

### ▣ `datetime.now(tz=None)`

- Return the current local date and time.
- If optional argument tz is None or not specified, this is like `today()`, but, if possible, supplies more precision than can be gotten from going through a time.

### ▣ `datetime.utcnow()`

- Return the current UTC date and time, with tzinfo None.

# 2 datetime Objects

## □ Class attributes

### ▣ **datetime.min**

- The earliest representable datetime
- `datetime(MINYEAR, 1, 1, tzinfo=None)`

### ▣ **datetime.max**

- The latest representable datetime
- `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`

### ▣ **datetime.resolution**

- The smallest possible difference between non-equal datetime objects
- `timedelta(microseconds=1)`



# 2 datetime Objects

- Instance attributes (read-only)
  - ▣ **datetime.year**
    - Between MINYEAR and MAXYEAR inclusive
  - ▣ **datetime.month**
    - Between 1 and 12 inclusive.
  - ▣ **datetime.day**
    - Between 1 and the number of days in the given month of the given year

# 2 datetime Objects

- Instance attributes (read-only)
  - ▣ **datetime.hour**
    - In range(24)
  - ▣ **datetime.minute**
    - In range(60)
  - ▣ **datetime.second**
    - In range(60)
  - ▣ **datetime.microsecond**
    - In range(1000000)

# 2 datetime Objects

- Instance methods

- ▣ `datetime.isoweekday()`

- Return the day of the week as an integer, where Monday is 1 and Sunday is 7.

# 2 datetime Objects

## □ Instance methods

### ▣ `datetime.isoformat(sep='T', timespec='auto')`

- Return a string representing the date and time in ISO 8601 format, YYYY-MMDDTHH:MM:SS.
- mmmmmm or, if microsecond is 0, YYYY-MM-DDTHH:MM:SS.
- The optional argument `sep (default 'T')` is a one-character separator, placed between the date and time portions of the result.

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
>>>     def utcoffset(self, dt): return timedelta(minutes=-399)

>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

# 2 datetime Objects

## □ Instance methods

### ▣ `datetime.isoformat(sep='T', timespec='auto')`

- The optional argument `timespec` specifies the number of additional components of the time to include.
- It can be one of the following
  - `'auto'`: Same as `'seconds'` if microsecond is 0, same as `'microseconds'` otherwise.
  - `'hours'`: Include the hour in the two-digit HH format.
  - `'minutes'`: Include hour and minute in HH:MM format.
  - `'seconds'`: Include hour, minute, and second in HH:MM:SS format.
  - `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
  - `'microseconds'`: Include full time in HH:MM:SS.mmmmmm format.

# 2 datetime Objects

## □ Instance methods

### ▣ `datetime.strftime(format)`

- Return a string representing the date and time, controlled by an explicit format string.
- The following is a list of some of the format codes

# 2 datetime Objects

Directive	Meaning	Example
%a	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat(en_US) So, Mo, ..., Sa (de_DE)
%A	Weekday as locale's full name.	Sunday, Monday, ...,Saturday (en_US) Sonntag, Montag, ...,Samstag (de_DE)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6
%d	Day of the month as a zero-padded decimal number.	01, 02, ..., 31
%b	Month as locale's abbreviated name.	Jan, Feb, ..., Dec(en_US) Jan, Feb, ..., Dez(de_DE)
%B	Month as locale's full name.	January, February, ...,December (en_US) Januar, Februar, ...,Dezember (de_DE)
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99
%Y	Year with century as a decimal number.	0001, 0002, ..., 2013,2014, ..., 9998, 9999

# 2 datetime Objects

Directive	Meaning	Example
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US);      am, pm (de_DE)
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%S	Second as a zero-padded decimal number.	00, 01, ..., 59



# 2 datetime Objects

- Example

- ▣ Seven-segment Indicator

- (DrawSevenSegDisplay)(DrawSevenSegDisplay2)