Computer Science 1000: Part #7

Programming in Python

PROGRAMMING LANGUAGES: AN OVERVIEW

THE PYTHON PROGRAMMING LANGUAGE

IMPLEMENTING PROGRAMMING

Programming Languages: An Overview

- Disadvantages of assembly language:
  1. Low-level / concrete conception of data, e.g., numbers, registers $\Longleftrightarrow$ memory.
  2. Low-level / concrete conception of task, e.g., `ADD`, `COMPARE`, `JUMP`.
  3. Machine-specific.
  4. Not like natural language.
- Advantages of high-level programming language:
  1. High-level / abstract conception of data, e.g., lists, data item $\Longleftrightarrow$ data item.
  2. High-level / abstract conception of task, e.g., `IF-THEN-ELSE`, `WHILE` loop.
  3. Machine-independent*.
  4. Like natural language.

Programming Languages: An Overview (Cont'd)

- A programming language is defined by the valid statements in that language (**syntax**) and what those statements do (**semantics**).
- A programming language can be **compiled** (whole program translated into machine language) or **interpreted** (individual program-statements translated as needed).
- Machine-independence achieved formally by standards, e.g., ANSI, IEEE, and implemented in practice by intermediate languages, e.g., **bytecode**.
- Machine-independence is often violated, e.g., may exploit particular machines and/or modify language features; additional incompatible variants may arise as language evolves over time, e.g., Python 2.x vs. Python 3.x.

Programming Languages: An Overview (Cont'd)

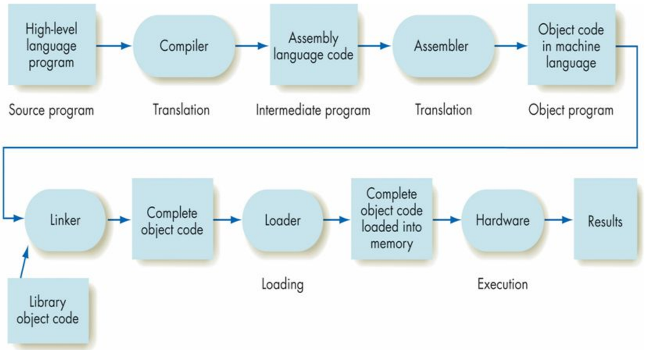


Figure 8.1
Transitions of a High-level Language Program

Programming Languages: An Overview (Cont'd)

Two reasons why there are many programming languages:

1. Languages are designed for different tasks, e.g.,
    - Scientific computation (FORTRAN)
    - Business applications (COBOL)
    - Web-page creation (HTML)
    - Database creation (SQL)
2. Languages are designed for different ways of thinking about programming, e.g.,
    - Procedural programming (FORTRAN, COBOL, C)
    - Object-oriented programming (OOP) (C++, Java)
    - Logic Programming (Prolog)
    - Script-based programming (Javascript, Ruby)

The Python Programming Language: Overview

- Created by Guido van Rossum in 1991 as an easy-to-learn general-purpose programming language.
- Procedural scripting language that allows but does not require OOP (“as OOP as you wanna be”).
- Key design principles:
  - Control structure indicated by indentation.
  - Powerful built-in data types.
  - Any variable can refer to any type of data, and this type can change as a program executes.
- Primarily interpreted but can be compiled for speed.
- General machine-independence achieved by bytecode; however, Python 3.x not directly backward-compatible with Python 2.x.

The Python Programming Language: A First Example Program

```
# Example program; adapted from
# Online Python Supplement, Figure 1.2

speed = input("Enter speed (mph):  ")
speed = int(speed)
distance = input("Enter distance (miles):  ")
distance = float(distance)

time = distance / speed

print("At", speed, "mph, it will take")
print(time, "hours to travel", \
      distance, "miles.")
```

The Python Programming Language: A First Example Program (Cont'd)

- Python programs are stored in files with extension `.py`, e.g., `example1.py`.
- When this program is executed using a Python interpreter and the user enters the boldfaced values, this is printed:

```
Enter speed (mph):  58
Enter distance (miles):  657.5
At 58 mph it will take
11.3362068966 hours to travel 657.5 miles.
```

The Python Programming Language: A First Example Program (Cont'd)

- Line numbers not necessary; are given here to allow easy reference to program lines.
- Lines beginning with hash (#) are **comments** (Lines 1-2); a **prologue comment** at the top of the program gives a program's purpose and creation / modification history.
- Comment and blank lines (Lines 3, 8, and 10) are ignored.
- Each line is a program statement; multiline statements are linked by end-of-line backslashes (\) (Lines 12-13).
- No variable-type declaration statements; this is handled by **assignment statements** (Lines 4-7 and 9).
- This program also has basic **I/O statements** (Lines 4, 6, and 11-13); **control statements** will be shown later.

The Python Programming Language: Assignment Statements

- General form: *variable = expression*, e.g.,
  - `index = 1`
  - `myDistanceRate = curDistanceRate * 1.75`
  - `name = "Todd Wareham"`
  - `curDataFilename = main + ".txt"`
  - `callList = ["Bob", "Sue", "Anne"]`
- Sets the value of *variable* to the value of *expression*.
  - If *variable* did not already exist, it is created.
  - If *variable* did already exist, its previous value is replaced. Note that the data-type of this previous value need not be that of the value created by *expression*.

The Python Programming Language: Assignment Statements (Cont'd)

- Variable names (also called **identifiers**) can be arbitrary sequences of letters, numbers and underscore symbols (_) such that (1) the first symbol is a letter and (2) the sequence is not already used in the Python language, e.g., `if`, `while`.
- Python is case-sensitive wrt letter capitalization, e.g., `myList` is a different variable than `mylist`.
- By convention, variables are a mix of lower- and upper-case letters and numbers; words may be combined to form a variable name in so-called "camel-style", e.g., `myList`, `dataFilename1`.

The Python Programming Language: Assignment Statements (Cont'd)

- By convention, constants use only upper-case letters and numbers, e.g., PI, TYPE1COLOR.
  - Though constants should not change value, they are still technically variables, e.g.,

```
...
PI = 3.1415927
...
PI = -1
...
```

    It is up to programmers to make sure that such changes do not happen.

- Underscores reserved for Python system constants.

The Python Programming Language: Assignment Statements (Cont'd)

- The `int` and `float` data-types
  - Encode "arbitrary" integers, e.g., `-1001, 0, 57`, and floating-point numbers, e.g. `-100.2, 3.1415927`.
  - Support basic arithmetic operations (`+, -, *, /`); also have floor-division (`//`) and remainder (`%`) operations, e.g.,

`7 / 2`	$\Longrightarrow$	3.5
`7 // 2`	$\Longrightarrow$	3
`7 % 2`	$\Longrightarrow$	1

    Behaviour of `/` incompatible with Python 2.x.
  - Many additional math functions and constants available in the `math` module, e.g., `abs(x)`, `pow(base, exponent)`, `sqrt(x)`, `pi`.

The Python Programming Language: Assignment Statements (Cont'd)

```
radius = input("Enter radius:  ")
radius = float(radius)
area = 3.1415927 * radius * radius
print("Circle Area = ", area)
```

```
import math

radius = input("Enter radius:  ")
radius = float(radius)
area = math.pi * math.pow(radius, 2)
print("Circle Area = ", area)
```

The Python Programming Language: Assignment Statements (Cont'd)

- The `str` data-type
  - Encodes "arbitrary" character strings, e.g., `"657.5"`, `"Todd Wareham"`.
  - Supports many operations, e.g.,
    - Concatenation (+) (`"Todd" + " " + "Wareham"` $\Longrightarrow$ `"Todd Wareham"`)
    - Lower-casing ("Todd".lower() $\Longrightarrow$ "todd")
    - Upper-casing ("Todd".upper() $\Longrightarrow$ "TODD")
- Convert between data types using **type casting** functions, e.g., `float("657.5")` $\Longrightarrow$ `657.5`, `int(657.5)` $\Longrightarrow$ `657`, `str(58)` $\Longrightarrow$ `"58"`.

The Python Programming Language: Assignment Statements (Cont'd)

- The `list` data-type
  - Encodes "arbitrary" lists, e.g., `[22, 5, 13, 57, -1]`, `["Bob", "Sue", "Anne"]`.
  - Items in list `L` indexed from 0 as `L[IND]`, e.g., if `L = [22, 5, 13, 57, -1]`, `L[0]` $\Longrightarrow$ `22` and `L[4]` $\Longrightarrow$ `-1`.
  - Supports many operations, e.g.,
    - Number of values in list (`len(L)`)
    - Append x to right end of list (`L.append(x)`)
    - List sorting (`L.sort()`)
    - Get list maximum value (`max(L)`)

The Python Programming Language: I/O Statements

- Keyboard input done via `input(string)`.
  - Prints `string` on screen, waits for user to enter input followed by a key return, and then returns this input-string.
  - Input-string can be converted as necessary by type-casting functions, e.g., `float(radius)`.
- Screen output done via `print(plist)`.
  - Comma-separated items in `plist` converted to strings as necessary and concatenated, and resulting string printed.
  - By default, each `print`-statement prints one line; can override this by making `end = " ")` the last item.
  - Can include escape characters to modify printout, e.g., `\t` (tab), `\n` (newline),
- Above I/O incompatible with Python 2.x.

The Python Programming Language: I/O Statements (Cont'd)

The statements

```
print("Here is \t a weird")
print("way \n of printing ", end = " ")
print("this message.")
```

print(out)

```
Here is      a weird
way
of printing this message.
```

The Python Programming Language: A First Example Program Redux

```
# Example program; adapted from
# Online Python Supplement, Figure 1.2

speed = input("Enter speed (mph):  ")
speed = int(speed)
distance = input("Enter distance (miles):  ")
distance = float(distance)

time = distance / speed

print("At", speed, "mph, it will take")
print(time, "hours to travel", \
      distance, "miles.")
```

The Python Programming Language: Control Statements

- Sequential Statements (**Statement Block**):
  - A set of statements with the same indentation.
  - All Python programs seen so far are purely sequential.
- Conditional Statements:
  - General form:

    `if (`**CONDITION1**`):`
    $\langle$ **CONDITION1 Block** $\rangle$
    `elif (`**CONDITION2**`):`
    $\langle$ **CONDITION2 Block** $\rangle$
    ...
    `else:`
    $\langle$ **ELSE Block** $\rangle$
  - `elif` and `else` blocks are optional.

The Python Programming Language: Control Statements (Cont'd)

Conditions typically based on variable-comparisons, possibly connected together by logical operators.

`x == y`	x equal to y
`x != y`	x not equal to y
`x < y`	x less than y
`x <= y`	x less than or equal to y
`x > y`	x greater than y
`x >= y`	x greater than or equal to y
`E1 and E2`	logical AND of $E1$ and $E2$
`E1 or E2`	logical OR of $E1$ and $E2$
`not E1`	logical NOT of $E1$

The Python Programming Language: Control Statements (Cont'd)

```
if ((number % 2) == 0):
    print("number is even")
```

```
if ((number >= 1) and (number <= 10)):
    print("number in range")
```

```
if (1 <= number <= 10)):
    print("number in range")
```

```
if not (1 <= number <= 10)):
    print("number not in range")
```

The Python Programming Language: Control Statements (Cont'd)

```
if ((number % 2) == 0):
    print("number is even")
else:
    print("number is odd")
```

```
if (number < 10):
    print("number less than 10")
elif (number == 10):
    print("number equal to 10")
else:
    print("number greater than 10")
```

The Python Programming Language: Control Statements (Cont'd)

- Conditional Looping Statement:
  - General form:

    `while (`**CONDITION**`):`

    ⟨ **Loop Block** ⟩

  - Executes **Loop Block** as long as **CONDITION** is True.
- Iterated Looping Statement:
  - General form:

    `for x in `**LIST**`:`

    ⟨ **Loop Block** ⟩

  - Executes **Loop Block** for each item x in LIST.

The Python Programming Language: Control Statements (Cont'd)

Print the numbers between 1 and 100 inclusive:

```
number = 1
while (number <= 100):
    print(number)
    number = number + 1
```

```
for number in range(1, 101):
    print(number)
```

The Python Programming Language: Control Statements (Cont'd)

Sum the numbers in a -1-terminated list:

```
sum = 0
number = int(input("Enter number:  "))
while (number != -1):
   sum = num + number
   number = int(input("Enter number:  "))
print("Sum is ", sum)
```

The Python Programming Language: Control Statements (Cont'd)

Find the maximum value in a -1-terminated list:

```
maxValue = -99
number = int(input("Enter number:  "))
while (number != -1):
   if (number > maxValue):
      maxValue = number
   number = int(input("Enter number:  "))
print("Maximum value is ", maxValue)
```

The Python Programming Language: Control Statements (Cont'd)

Store the values in a -1-terminated list in L:

```
L = []
number = int(input("Enter number:  "))
while (number != -1):
   L.append(number)
   number = int(input("Enter number:  "))
```

Print the values in list L (one per line):

```
for number in L:
   print(number)
```

The Python Programming Language: Control Statements (Cont'd)

Sort the n values in list L (Selection Sort pseudocode):

ENDUNSORT = n
While (ENDUNSORT $>$ 1) do
    FPOS = 1
    for IND = 2 to ENDUNSORT do
        If $L_{IND} > L_{FPOS}$ then
            FPOS = IND
    TMP = $L_{ENDUNSORT}$
    $L_{ENDUNSORT}$ = L_{FPOS}
    L_{FPOS} = TMP
    ENDUNSORT = ENDUNSORT - 1

The Python Programming Language: Control Statements (Cont'd)

Sort the values in list L (Selection Sort):

```
endUnSort = len(L) - 1
while (endUnSort > 0):
   maxPos = 0
   for ind in range(1, endUnSort + 1):
      if (L[ind] > L[maxPos]):
        maxPos = ind
   tmp = L[endUnSort]
   L[endUnSort] = L[maxPos]
   L[maxPos] = tmp
   endUnSort = endUnSort - 1
```

The Python Programming Language: Control Statements (Cont'd)

Store unique values in sorted list L in list $LUnique$:

```
LUnique = []
curValue = L[0]
for ind in range(1, len(L)):
   if (L[ind] != curValue):
      LUnique.append(curValue)
      curValue = L[ind]
LUnique.append(curValue)
```

The Python Programming Language: Functions

- Compartmentalize data and tasks in programs with **functions**; allow implementation of divide-and-conquer-style programming.
- General form:

```
def funcName():
    ⟨ Function Block ⟩

def funcName(parameterList):
    ⟨ Function Block ⟩

def funcName(parameterList):
    ⟨ Function Block ⟩
    return value
```

The Python Programming Language: Functions (Cont'd)

- A variable defined inside a function is a **local variable**; otherwise, it is a **global variable**.
- If a local variable has the same name as a global variable, the local variable is used inside the function.
- What does this print?

```
def myFunc1():
    one = -1
    print(one, two)

one = 1
two = 2
print(one, two)
myFunc1()
print(one, two)
```

The Python Programming Language: Functions (Cont'd)

- The parameters in a function's parameter-list match up with and get their values from the arguments in the argument-list of a function call in numerical order, not by parameter / argument name.
- What does this print?

```
def myFunc2(one, two, three):
    print(one, two, three)

one = 1
two = 2
three = 3
print(one, two, three)
myFunc2(two, three, one)
print(one, two, three)
```

The Python Programming Language: Functions (Cont'd)

- The value returned by a function can be captured by an assignment statement which has that function as the expression.
- What does this print?

```
def myFunc3(one, two, three):
    sum = (one + two) - three
    return sum

one = 1
two = 2
three = 3
result = myFunc3(two, three, one)
print(result)
```

The Python Programming Language: Functions (Cont'd)

- Eliminate global variables with **main functions.**
- What does this print?

```
def myFunc4(one, two, three):
    sum = (one + two) - three
    return sum

def main():
    one = 1
    two = 2
    three = 3
    result = myFunc4(two, three, one)
    print(result)

main()
```

The Python Programming Language: Functions (Cont'd)

- Compartmentalize data and tasks in programs with functions; allow implementation of **divide-and-conquer-style programming** (which is based on the levels-of-abstraction organizational principle).
- Functions useful in all stages of software development:
  1. Planning (View complex problem as set of simple subtasks)
  2. Coding (Code individual subtasks independently)
  3. Testing (Test individual subtasks independently)
  4. Modifying (Restrict changes to individual subtasks)
  5. Reading (Understand complex problem as set of simple subtasks)

The Python Programming Language: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #1):

```
L = []
number = int(input("Enter number:  "))
while (number != -1):
   L.append(number)
   number = int(input("Enter number:  "))
for number in L:
   print(number)
```

The Python Programming Language: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #2):

```
def readList():
    L = []
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))

def printList():
    for number in L:
        print(number)

readList()
printList()
```

The Python Programming Language: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #3):

```
def readList():
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))

def printList():
    for number in L:
        print(number)

L = []
readList()
printList()
```

The Python Programming Language: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #4):

```
def readList():
    L = []
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))
    return L

def printList(L):
    for number in L:
        print(number)

L = readList()
printList(L)
```

The Python Programming Language: Functions (Cont'd)

```
def readList():
    L = []
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))
    return L

def printList(L):
    for number in L:
        print(number)

def main():
    L = readList()
    printList(L)

main()
```

The Python Programming Language: Functions (Cont'd)

Sort the values in list L (Selection Sort) (Function):

```
def sortList(L):
    endUnSort = len(L) - 1
    while (endUnSort > 0):
        maxPos = 0
        for ind in range(1, endUnSort + 1):
            if (L[ind] > L[maxPos]):
                maxPos = ind
        tmp = L[endUnSort]
        L[endUnSort] = L[maxPos]
        L[maxPos] = tmp
        endUnSort = endUnSort - 1
    return L
```

The Python Programming Language: Functions (Cont'd)

Compute unique values in sorted list L (Function):

```
def getUniqueList(L):
    LUnique = []
    curValue = L[0]
    for ind in range(1, len(L)):
        if (L[ind] != curValue):
            LUnique.append(curValue)
            curValue = L[ind]
    LUnique.append(curValue)
    return LUnique
```

The Python Programming Language: Functions (Cont'd)

Main function for unique-value list program:

```
def main():
    L = readList()
    L = sortList(L)
    L = getUniqueList(L)
    printList(L)
```

The Python Programming Language: Object-Oriented Programming

- Python implements OOP using standard dot syntax, e.g.,
  - `object.attribute` (internal object attribute, e.g., `c.radius`)
  - `object.function(plist)` (internal object function, e.g., `L.sort()`)

  Note that some attributes / functions are publicly available and others are private to the object itself.
- Objects typically created by assignment statements in which expression is special object-constructor function, e.g., `o = object(plist)`.
- Illustrate OOP via graphics library (Dr. John Zelle).

The Python Programming Language: Graphics Programming

- Graphics critical in GUI and visualization.
- Graphics screen hardware is **bitmapped display** $(1560 \times 1280$ pixels); by convention, position $(0, 0)$ is in the upper lefthand corner.
- Each pixel in this display directly maps to one element of the **frame buffer** $(1560 \times 1280 \times 24$ bits / pixel $=$ 6 MB).
- Due to screen fading, **each** pixel re-painted / refreshed on screen 30–50 times pers second to avoid flicker.
- Objects in Python graphics library model not only graphics window on screen but also all high-level graphics objects that are displayed in that window.
- Invoke library via command `from graphics import *`.

The Python Programming Language: Graphics Programming (Cont'd)

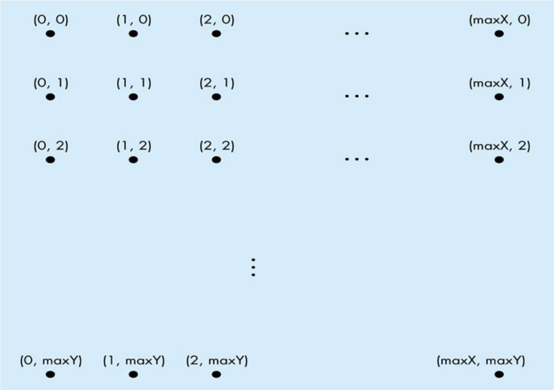


Figure 8.34

Pixel Numbering System in a Bitmapped Display

The Python Programming Language: Graphics Programming (Cont'd)

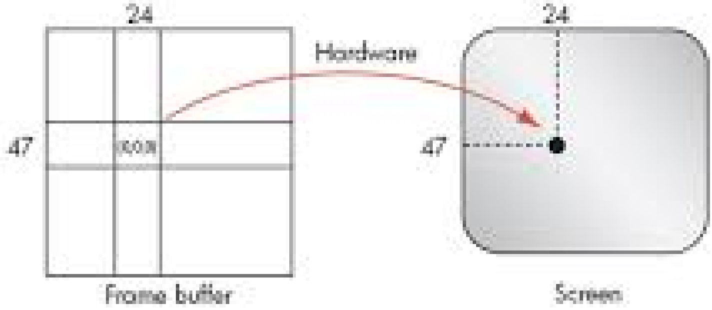
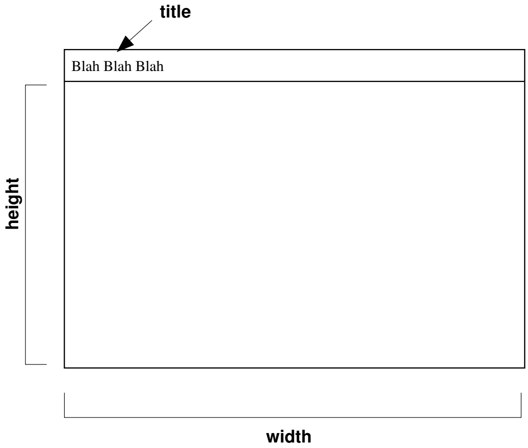


Figure 33 Display of Information on the Terminal

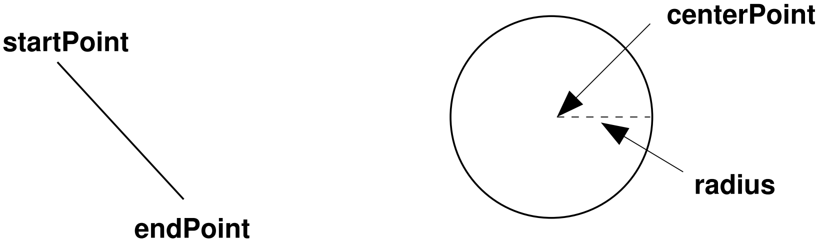
The Python Programming Language: Graphics Programming (Cont'd)

```
win = GraphWin(title, width, height)
```



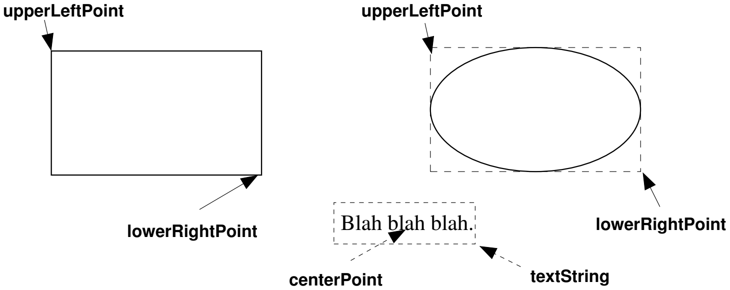
The Python Programming Language: Graphics Programming (Cont'd)

```
point = Point(x, y)
line = Line(startPoint, endPoint)
circle = Circle(centerPoint, radius)
```



The Python Programming Language: Graphics Programming (Cont'd)

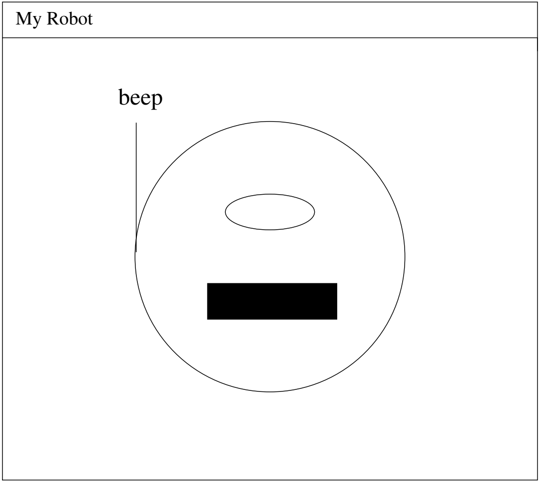
```
rect = Rectangle(upperLeftP, lowerRightP)
oval = Oval(upperLeftP, lowerRightP)
text = Text(centerP, textString)
```



The Python Programming Language: Graphics Programming (Cont'd)

```
from graphics import *

win = GraphWin("My Robot", 120, 100)
face = Circle(Point(60, 50), 30)
face.draw(win)
mouth = Rectangle(Point(45,55), Point(75,65))
mouth.setFill('black')
mouth.draw(win)
antenna = Line(Point(30,50), Point(30,20))
antenna.draw(win)
antennaText = Text(Point(30, 15), "beep")
antennaText.draw(win)
eye = Oval(Point(50,35), Point(70,45))
eye.draw(win)

win.getMouse()
win.close()
```

The Python Programming Language: Graphics Programming (Cont'd)



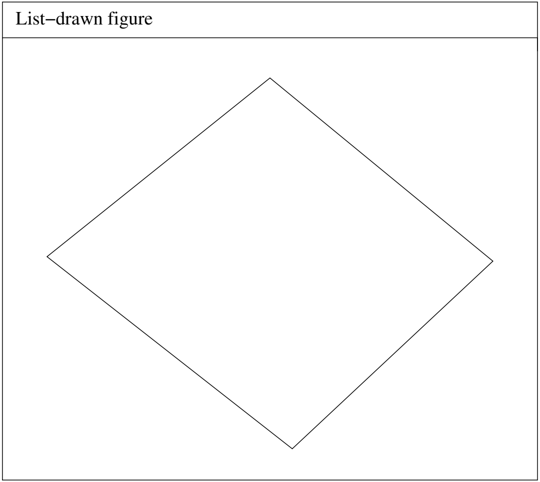
The Python Programming Language: Graphics Programming (Cont'd)

- To draw graphics-object o in graphics window win, use command `o.draw(win)`.
- To color interior of circle, rectangle, or oval graphics object o, use command `o.setFill(colorString)`, e.g., `rect.setFill('blue')`.
- Make sure all drawn lines are inside the grid defined on the graphics window – otherwise, portions of what you want to draw will be missing ("If it's not in the frame, it doesn't exist." – *Shadow of the Vampire* (2000)).
- An alternative to drawing lines object by object is to create a list of line-objects and then draw them using a `for`- loop.

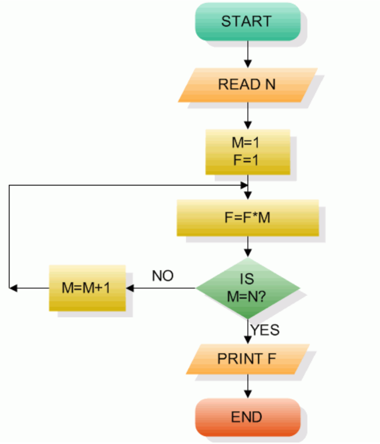
The Python Programming Language: Graphics Programming (Cont'd)

```
from graphics import *

win = GraphWin("List-drawn figure", 120, 100)
L = []
L.append(Line(Point(60,10), Point(110,50)))
L.append(Line(Point(110,50), Point(60,90)))
L.append(Line(Point(60,90), Point(10,50)))
L.append(Line(Point(10,50), Point(60,10)))
for line in L:
    line.draw(win)

win.getMouse()
win.close()
```

The Python Programming Language: Graphics Programming (Cont'd)

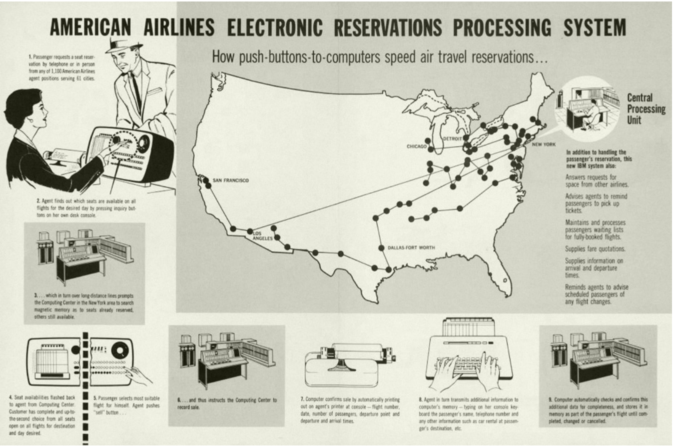


Implementing Programming: The Software Crisis



- Act of programming made easier by compilers, languages, and operating systems; problem of developing algorithms remained.
- Special notations like flowcharts help with small- and medium-size programs; hope was that appropriate management would help with large ones.

Implementing Programming: The Software Crisis (Cont'd)



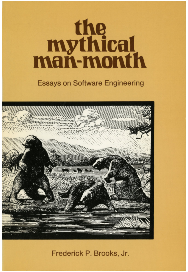
The SABRE Airline Reservation System (1964)

Implementing Programming: The Software Crisis (Cont'd)

IBM System/360 (1967)

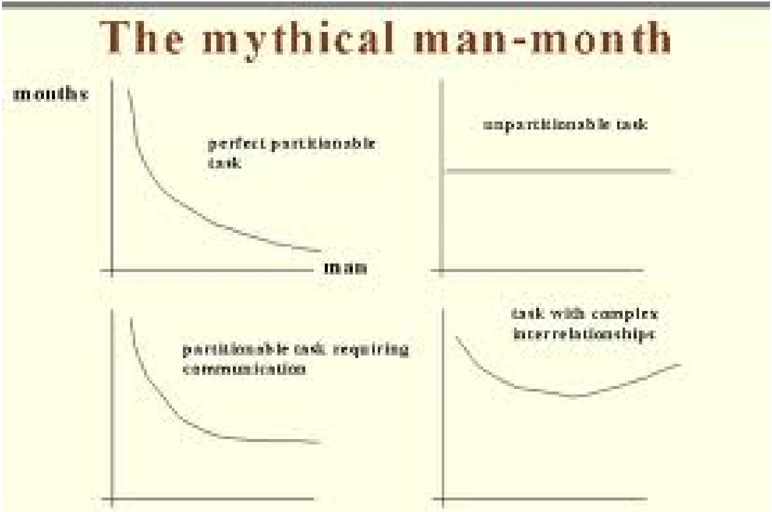
Implementing Programming: The Software Crisis (Cont'd)

Fred Brooks Jr. (1931–)

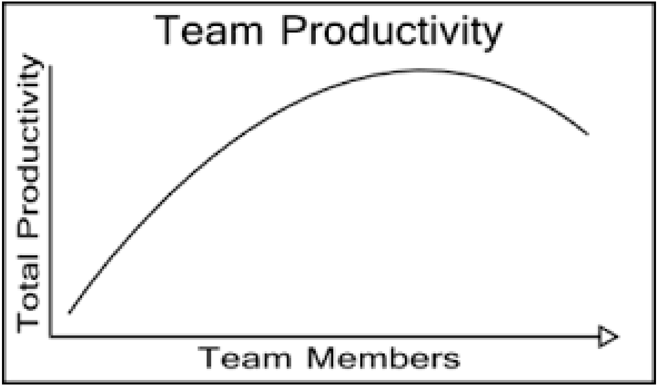


- OS/360 initially planned for 1965 costing $125M; limped to market in 1967 costing $500M, and virtually destroyed IBM's in-house programming division.
- Brooks discussed causes in *The Mythical Man Month*.

Implementing Programming: The Software Crisis (Cont'd)



Implementing Programming: The Software Crisis (Cont'd)



As both larger programs and larger teams have more complex internal relationships, adding more programmers to larger projects makes things *worse*.

Implementing Programming: The Software Crisis (Cont'd)

- Software Engineering born at 1968 NATO-sponsored conference; goal of SE is to develop efficient processes for creating and maintaining correct software systems.
- Many types of processes proposed, *e.g.*, design and management methodologies (Agile), automatic software derivation methods; however, "No Silver Bullet" (Brooks).

. . . And If You Liked This . . .

- MUN Computer Science courses on this area:
    - COMP 1001: Introduction to Programming
    - COMP 2001: Object-oriented Programming and HCI
    - COMP 2005: Software Engineering
    - COMP 4711: Structure of Programming Languages
- MUN Computer Science professors teaching courses / doing research in in this area:
    - Miklos Bartha
    - Ed Brown
    - Rod Byrne
    - Adrian Fiech