Last year, for the first time, spending by Apple and Google on patent lawsuits and unusually big-dollar patent purchases exceeded spending on research and development of new products

But in 2008, Mr. Phillips's company, Vlingo, had been contacted by a much larger voice recognition firm called Nuance. "I have patents that can prevent you from practicing in this market," Nuance's chief executive, Paul Ricci, told Mr. Phillips, according to executives involved in that conversation.

Mr. Ricci issued an ultimatum: Mr. Phillips could sell his firm to Mr. Ricci or be sued for patent infringements. When Mr. Phillips refused to sell, Mr. Ricci's company filed the first of six lawsuits.

Soon after, Apple and Google stopped returning phone calls. The company behind Siri switched its partnership from Mr. Phillips to Mr. Ricci's firm. And the millions of dollars Mr. Phillips had set aside for research and development were redirected to lawyers and court fees.

When the first lawsuit went to trial last year, Mr. Phillips won. In the companies' only courtroom face-off, a jury ruled that Mr. Phillips had not infringed on a broad voice recognition patent owned by Mr. Ricci's company.

But it was too late. The suit had cost $3 million, and the financial damage was done. In December, Mr. Phillips agreed to sell his company to Mr. Ricci. "We were on the brink of changing the world before we got stuck in this legal muck," Mr. Phillips said.

http://www.nytimes.com/2012/10/08/technology/patent-wars-among-tech-giants-can-stifle-competition.html

- Should be done grading next Tuesday

- Lab 5: due at 11:55pm on Friday
- Lab 6: starting today in lab

import graphics
from graphics import *
only need one, at top, below comment

main should be at bottom (but still works at top)

whitespace between functions

pop up window after input

close window at end

code outside functions: create window only once

String Formatting

Change Counter

Please enter the count of each coin type.

Quarters: 6

Dimes: 0

Nickels: 0

Pennies: 0

The total value of your change is 1.5

- Shouldn't that be more like $1.50!?

String Formatting

- We can format our output by modifying the print statement as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

The total value of your change is $1.50

- Key: use the string `format` method

String Formatting

- `<template-string>.format(<values>)`

- {} within the template-string mark "slots" into which the values are inserted

- Each slot has description that includes format specifier telling Python how the value for the slot should appear

String Formatting

```
print("The total value of your change is ${0:0.2f}".format(total)
```

- The template contains a single slot with the description: `0:0.2f`
- Form of description: `<index>:<format-specifier>`

- Index tells which parameter to insert into the slot. In this case, total

String Formatting

- The formatting specifier has the form: `<width>.<precision><type>`

- **f** means "fixed point" number
- **<width>** tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- **<precision>** is the number of decimal places.

String Formatting

```
>>> "Hello {0} {1}, you may have won ${2}" .format("Mr.", "Smith", 10000)
'Hello Mr. Smith, you may have won $10000'

>>> 'This int, {0:5}, was placed in a field of width 5'.format(7)
'This int,     7, was placed in a field of width 5'

>>> 'This int, {0:10}, was placed in a field of witdh 10'.format(10)
'This int,         10, was placed in a field of witdh 10'

>>> 'This float, {0:10.5}, has width 10 and precision 5.'.format(3.1415926)
'This float,    3.1416, has width 10 and precision 5.'

>>> 'This float, {0:10.5f},  is fixed at 5 decimal places.'.format(3.1415926)
'This float,   3.14159, has width 0 and precision 5.'
```

String Formatting

- If the width is wider than needed, numeric values are right-justified and strings are left- justified, by default.
- You can also specify a justification before the width.

```
>>> "left justification: {0:<5}.format("Hi!")
'left justification: Hi!  '
>>> "right justification: {0:>5}.format("Hi!")
'right justification:  Hi!'
>>> "centered: {0:^5}".format("Hi!")
'centered: Hi! '
```

Aside

- Public / private key encryption
- RSA (1977)
  - Ron Rivest, Adi Shamir and Leonard Adleman
  - Factoring large integers

The Function of Functions

- Having similar or identical code in more than one place has some drawbacks
  - **Issue one**: writing the same code twice or more
  - **Issue two**: This same code must be maintained in two separate places
- Functions can be used to reduce code duplication and make programs more easily understood and maintained

Functions, Informally

- A function is like a ***subprogram***, a small program inside of a program
- The basic idea – we write a sequence of statements and then give that sequence a name. We can then execute this sequence at any time by referring to the name

Functions, Informally

- The part of the program that creates a function is called a *function definition*
- When the function is used in a program, we say the definition is called or *invoked*

Functions, Informally

- Happy Birthday lyrics…

```
def main():
    print("Happy birthday to you!" )
    print("Happy birthday to you!" )
    print("Happy birthday, dear Fred...")
    print("Happy birthday to you!")
```

- Gives us this…

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

Code reuse.

Functions, Informally

- Creating this function saved us a lot of typing!
- What if it's Lucy's birthday? We could write a new singLucy function

- ```
  def singLucy():
      happy()
      happy()
      print("Happy birthday, dear Lucy...")
      happy()
  ```
  ```

Functions, Informally

- We could write a main program to sing to both Lucy and Fred

```
def main():
    singFred()
    print()
    singLucy()
```

- This gives us this new output

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred..
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
Happy birthday to you!
```

Functions, Informally

- The generic function *sing*

```
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".“)
    happy()
```

- This function uses a parameter named person. A parameter is a variable that is initialized when the function is called

Functions, Informally

- Our new output –

```
>>> sing("Fred")
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

- We can put together a new main program

Functions, Informally

- Our new main program:

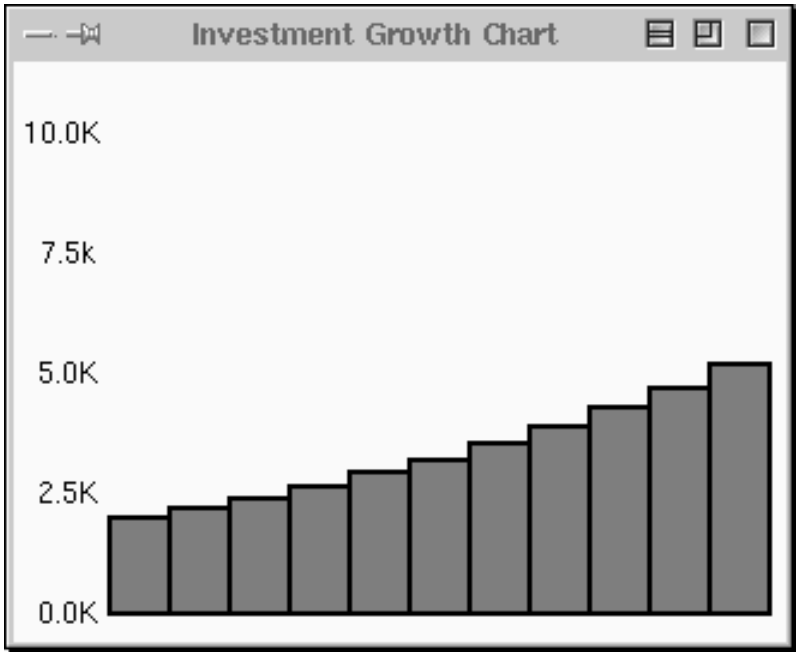
```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

- Gives us this output:

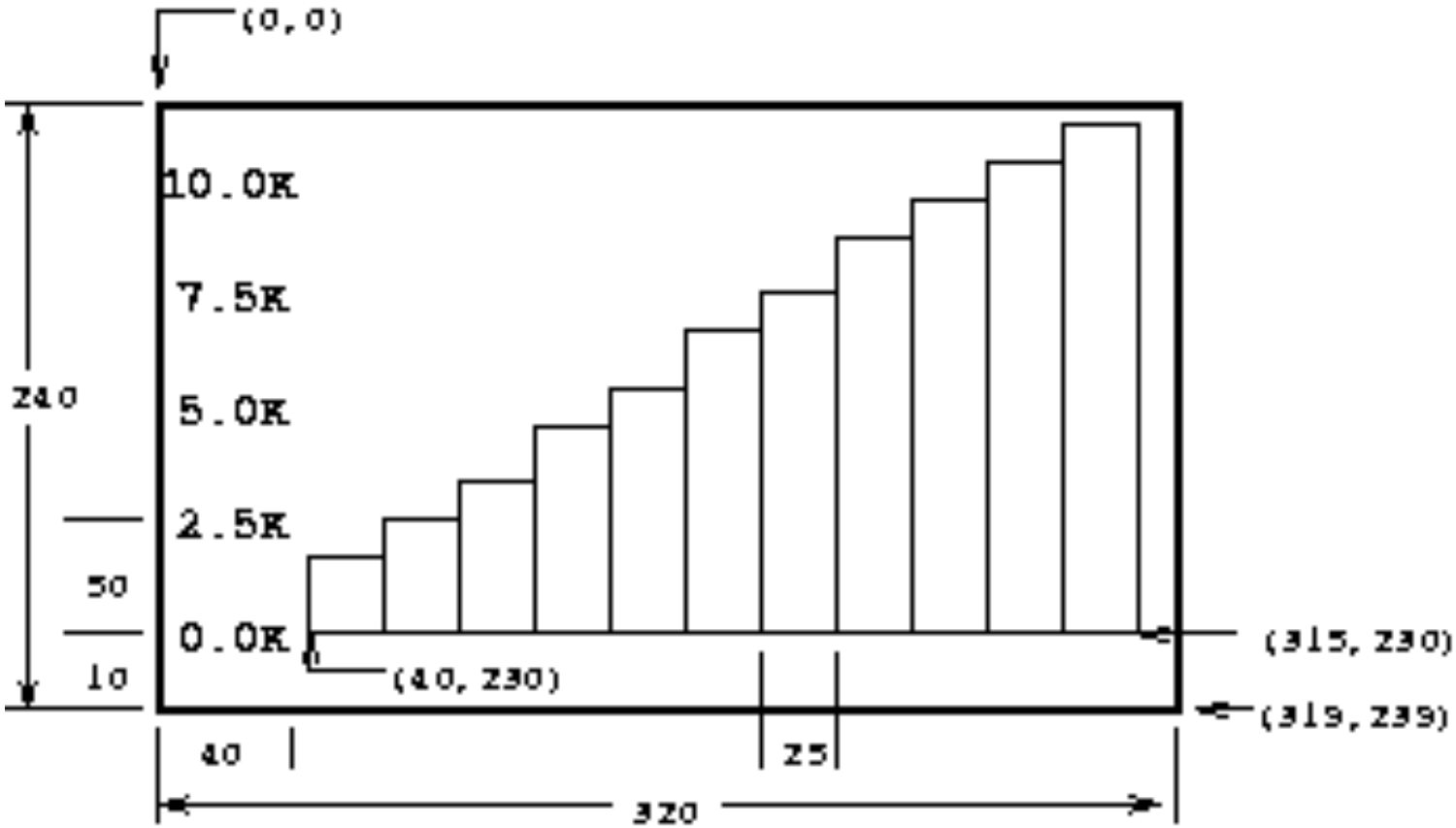
```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```

Graphing Future Value/ Choosing Coordinates



Graphing Future Value/ Choosing Coordinates



Future Value with a Function

- In the future value graphing program, we see similar code twice:

```
# Draw bar for initial principal
bar = Rectangle(Point(0, 0), Point(1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)


bar = Rectangle(Point(year, 0), Point(year+1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)
```

Future Value with a Function

- To properly draw the bars, we need three pieces of information.
  - The year the bar is for
  - How tall the bar should be
  - The window the bar will be drawn in
- These three values can be supplied as parameters to the function

Future Value with a Function

- The resulting function looks like this:

```
def drawBar(window, year, height):
   # Draw a bar in window starting at year with given height
   bar = Rectangle(Point(year, 0), Point(year+1, height))
   bar.setFill("green")
   bar.setWidth(2)
   bar.draw(window)
```

- To use this function, we supply the three values. If win is a Graphwin, we can draw a bar for year 0 and principal of $2000 using this call:

```
drawBar(win, 0, 2000)
```

Functions and Parameters: The Details

- It makes sense to include the year and the principal in the drawBar function, but why send the window variable?
- The scope of a variable refers to the places in a program a given variable can be referenced
- Each function is its own little subprogram
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter

Functions and Parameters: The Details

- Since the `GraphWin` in the variable `win` is created inside of `main`, it is not directly accessible in `drawBar`
- The `window` parameter in `drawBar` gets assigned the value of `win` from `main` when `drawBar` is called

Functions and Parameters: The Details

- A function definition looks like this:

```
def <name>(<formal-parameters>):
    <body>
```

- The name of the function must be an identifier
- Formal-parameters is a possibly empty list of variable names

Functions and Parameters: The Details

- Formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body

Functions and Parameters: The Details

- A function is called by using its name followed by a list of *actual parameters* or *arguments*

```
<name>(<actual-parameters>)
```

- When Python comes to a function call, it initiates a four-step process

Functions and Parameters: The Details

- Let's trace through the following code:

```
sing("Fred")
print()
sing("Lucy”)
```

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Fred"

```
def sing(person):
    happy()
    happy()
    print ("Happy birthday, dear", person + ".")
    happy()
```

person: "Fred"

Note that the variable `person` has just been initialized

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Fred"

```
def sing(person):
    happy()
    happy()
    print ("Happy birthday, dear", person + ".")
    happy()
```

```
def happy():
    print ("Happy Birthday to you!")
```

person: "Fred"

- Execution continues in this way with two more "trips" to `happy`
- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

```
def sing(person):
    happy()
    happy()
    print ("Happy birthday, dear", person + ".")
    happy()
```

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Lucy"

```
def sing(person):
    happy()
    happy()
    print ("Happy birthday, dear", person + ".")
    happy()
```

person: "Lucy"

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

```
def sing(person):
    happy()
    happy()
    print ("Happy birthday, dear", person + ".")
    happy()
```

Getting Results from a Function

- Passing parameters provides a mechanism for initializing the variables in a function
- Parameters act as inputs to a function
- We can call a function many times and get different results by changing its parameters

Functions That Return Values

- We've already seen numerous examples of functions that return values to the caller.

```
discRt = math.sqrt(b*b - 4*a*c)
```

- The value `b*b - 4*a*c` is the actual parameter of `math.sqrt`
- We say `sqrt` returns the square root of its argument

Functions That Return Values

- This function returns the square of a number:

```
def square(x):
   return x*x
```

- When Python encounters `return`, it exits the function and returns control to the point where the function was called
- In addition, the value(s) provided in the `return` statement are sent back to the caller as an expression result

Functions That Return Values

- Sometimes a function needs to return more than one value
- To do this, simply list more than one expression in the `return` statement

- def sumDiff(x, y):

```
    sum = x + y
    diff = x – y
    return sum, diff
```

Functions That Return Values

- When calling this function, use simultaneous assignment

```
num1, num2 = eval(input("Enter two numbers (num1, num2) "))
s, d       = sumDiff(num1, num2)
print("The sum is", s, "and the difference is", d)
```

- As before, the values are assigned based on position, so *s* gets the first value returned (the sum), and *d* gets the second (the difference)

Functions That Return Values

- One "gotcha" – all Python functions return a value, whether they contain a **`return`** statement or not. Functions without a **`return`** hand back a special object, denoted `None`
- A common problem is writing a value-returning function and omitting the `return`

Functions and Parameters: The Details

- As an example, consider the call to `drawBar`:
  `drawBar(win, 0, principal)`
- When control is passed to `drawBar`, these parameters are matched up to the formal parameters in the function heading:
  `def drawBar(window, year, height):`

Functions and Parameters: The Details

- The net effect is as if the function body had been prefaced with three assignment statements:

```
window = win
year = 0
height = principal
```

```
def foo(a, b):
    print("Foo", a, b)
    bar(a,b)

def bar(c, d):
    print("Bar", c, d)

def main():
    a = 2
    b = 3.0
    c = 'a'
    d = '4'

    print(a+b)
    print(c+d)
    print(str(a)+d)
    print(a+eval(d))

    foo(a,b)
    bar(b,d)
    print("Good\nBye")
```