Chapter Six
FUNCTIONS

CSC 161: The Art of Computer Programming

Matt Post (grad TA; guest lecturer)

10/26/2009

Course status

Course status

- Programming assignment 6: Graphics (part 2)
  - Due 10 AM October 31 (this Saturday)
  - To be completed in teams of two

Course status

- Programming assignment 6: Graphics (part 2)
  - Due 10 AM October 31 (this Saturday)
  - To be completed in teams of two
- Midterm exam
  - should have grades by this Wednesday

Course status

- Programming assignment 6: Graphics (part 2)
  - Due 10 AM October 31 (this Saturday)
  - To be completed in teams of two
- Midterm exam
  - should have grades by this Wednesday
- For Wednesday
  - make sure you've read Chapter 6

Head of an Old Peasant Woman. Pieter Brueghel the Elder c. 1565.

Monday, October 26, 2009

Imagine you're a peasant woman in 19th century England. Use this painting to seed your imagination (ignoring the fact that this is a 16th century peasant woman).

Making bread (ca. 200 years ago)

Monday, October 26, 2009

This is just part of your tedious quotidian routine: making bread. If only there was a way out of it.

Making bread (ca. 200 years ago)

- Wake up at 3 AM, put on bear skin or whatever

Monday, October 26, 2009

This is just part of your tedious quotidian routine: making bread. If only there was a way out of it.

Making bread (ca. 200 years ago)

- Wake up at 3 AM, put on bear skin or whatever
- Sift the bugs out of a few cups of flour, mix with some water and yeast and a little honey

Monday, October 26, 2009

This is just part of your tedious quotidian routine: making bread. If only there was a way out of it.

Making bread (ca. 200 years ago)

- Wake up at 3 AM, put on bear skin or whatever
- Sift the bugs out of a few cups of flour, mix with some water and yeast and a little honey
- Knead until you can no longer bear the pain in your wrists (you have carpal tunnel)

Monday, October 26, 2009

This is just part of your tedious quotidian routine: making bread. If only there was a way out of it.

Making bread (ca. 200 years ago)

- Wake up at 3 AM, put on bear skin or whatever
- Sift the bugs out of a few cups of flour, mix with some water and yeast and a little honey
- Knead until you can no longer bear the pain in your wrists (you have carpal tunnel)
- Set aside to rise; meanwhile, bring in some wood and use hot coals from yesterday to start a fire (walk to the neighbor's house if your coals died)

Monday, October 26, 2009

This is just part of your tedious quotidian routine: making bread. If only there was a way out of it.

Making bread (ca. 200 years ago)

- Wake up at 3 AM, put on bear skin or whatever
- Sift the bugs out of a few cups of flour, mix with some water and yeast and a little honey
- Knead until you can no longer bear the pain in your wrists (you have carpal tunnel)
- Set aside to rise; meanwhile, bring in some wood and use hot coals from yesterday to start a fire (walk to the neighbor's house if your coals died)
- et cetera ad nauseum

Monday, October 26, 2009

This is just part of your tedious quotidian routine: making bread. If only there was a way out of it.

Monday, October 26, 2009

There was a way out of it! If you were rich (like Queen Victoria), you could...

- Wake up at 3 AM, put on bear skin or whatever
- Sift the bugs out of a few cups of flour, mix with some water and yeast and a little honey
- Knead until you can no longer bear the pain in your wrists (you have carpal tunnel)
- Set aside to rise; meanwhile, bring in some wood and use hot coals from yesterday to start a fire (walk to the neighbor's house if your coals died)
- et cetera ad nauseum

http://tinyurl.com/yz4mphd

order a servant to do it!

- Wake up at 3 AM[illegible] whatever
- Sift the bugs out[illegible]r, mix with some water and yeast
- Knead until you[illegible]he pain in your wrists (you have[illegible]
- Set aside to rise[illegible]some wood and use hot coals from y[illegible]e (walk to the neighbor's hous[illegible]
- et cetera ad nauseum

Make bread!

http://tinyurl.com/yz4mphd

Monday, October 26, 2009

order a servant to do it!

http://tinyurl.com/yzn2quw

Monday, October 26, 2009

Instead of doing all those little jobs herself, the Queen just tells the servant to do it, and the servant delivers.

Make bread!

http://tinyurl.com/yzn2quw

Monday, October 26, 2009

Instead of doing all those little jobs herself, the Queen just tells the servant to do it, and the servant delivers.

Make bread!

http://tinyurl.com/yzn2quw

Monday, October 26, 2009

Instead of doing all those little jobs herself, the Queen just tells the servant to do it, and the servant delivers.

Extending the analogy

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it's behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it's behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen
  - can specify necessary or more detailed information

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it's behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen
  - can specify necessary or more detailed information
    - “Make rye bread!” make_bread(‘rye’)

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it’s behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen
  - can specify necessary or more detailed information
    - “Make rye bread!” make_bread(‘rye’)
    - “Make rye bread, quickly!” make_bread(‘rye’,’fast’)

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it’s behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen
  - can specify necessary or more detailed information
    - "Make rye bread!" make_bread('rye')
    - "Make rye bread, quickly!" make_bread('rye','fast')
  - does not know (or care) about the details of how the task is accomplished

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it's behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen
  - can specify necessary or more detailed information
    - "Make rye bread!" make_bread('rye')
    - "Make rye bread, quickly!" make_bread('rye','fast')
  - does not know (or care) about the details of how the task is accomplished
  - can ask for anything to be sent back

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it's behavior to vary slightly, and the receive the results of its work.

Extending the analogy

- The Queen
  - can specify necessary or more detailed information
    - "Make rye bread!" make_bread('rye')
    - "Make rye bread, quickly!" make_bread('rye','fast')
  - does not know (or care) about the details of how the task is accomplished
  - can ask for anything to be sent back
    - "Make tens loaves of rye!"
      loaves_list = make_bread('rye','fast',10)

Monday, October 26, 2009

Modern programming languages (like Python) make Queens of us all. Instead of painstakingly specifying the pieces of a task over and over again, we simply define a function, give it some parameters to allow it's behavior to vary slightly, and the receive the results of its work.

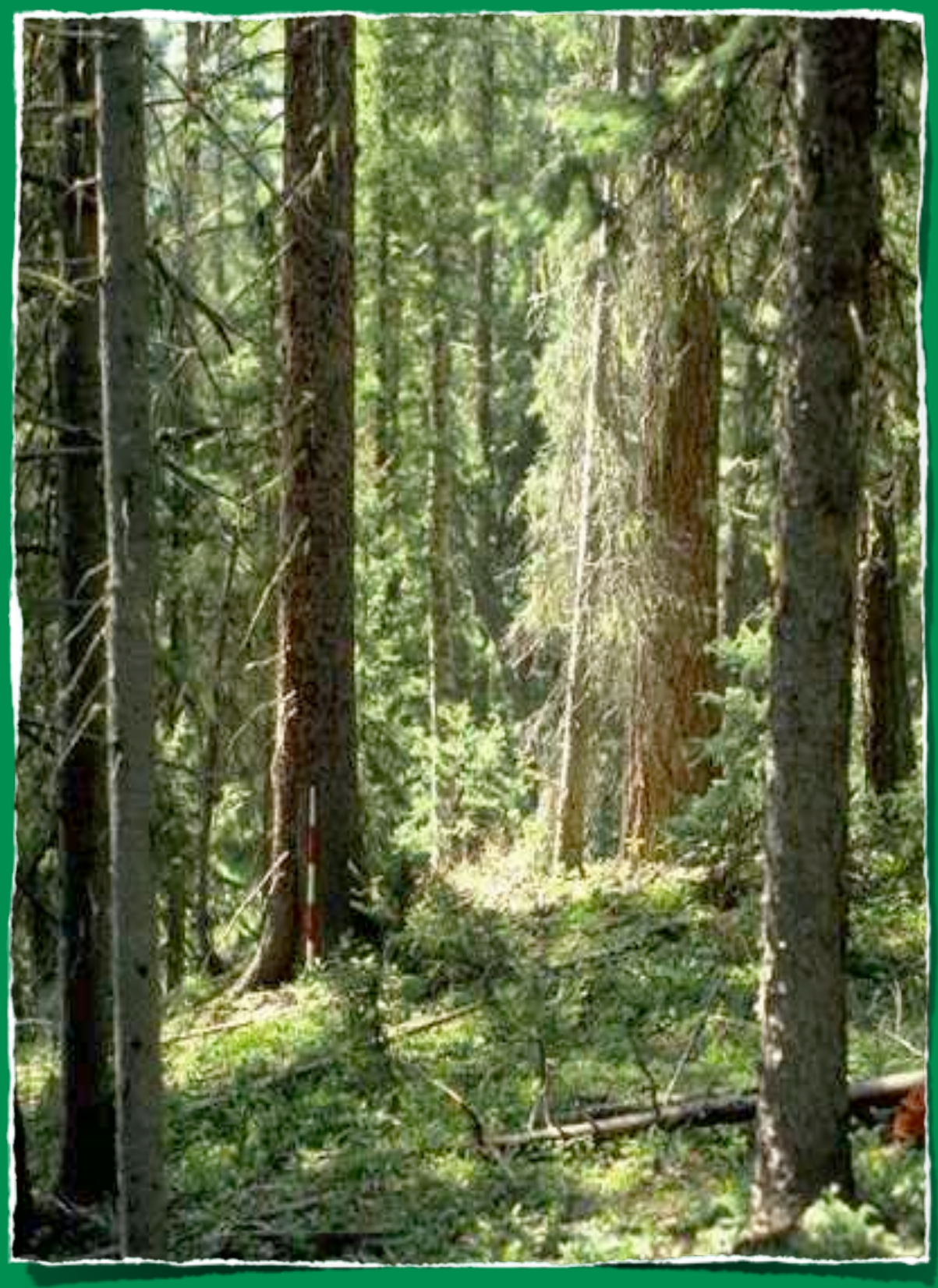
Why?

ELIMINATING REDUNDANCY

Monday, October 26, 2009

We now discuss details of how functions help eliminate wasteful and error-prone repetition.

eliminating redundancy

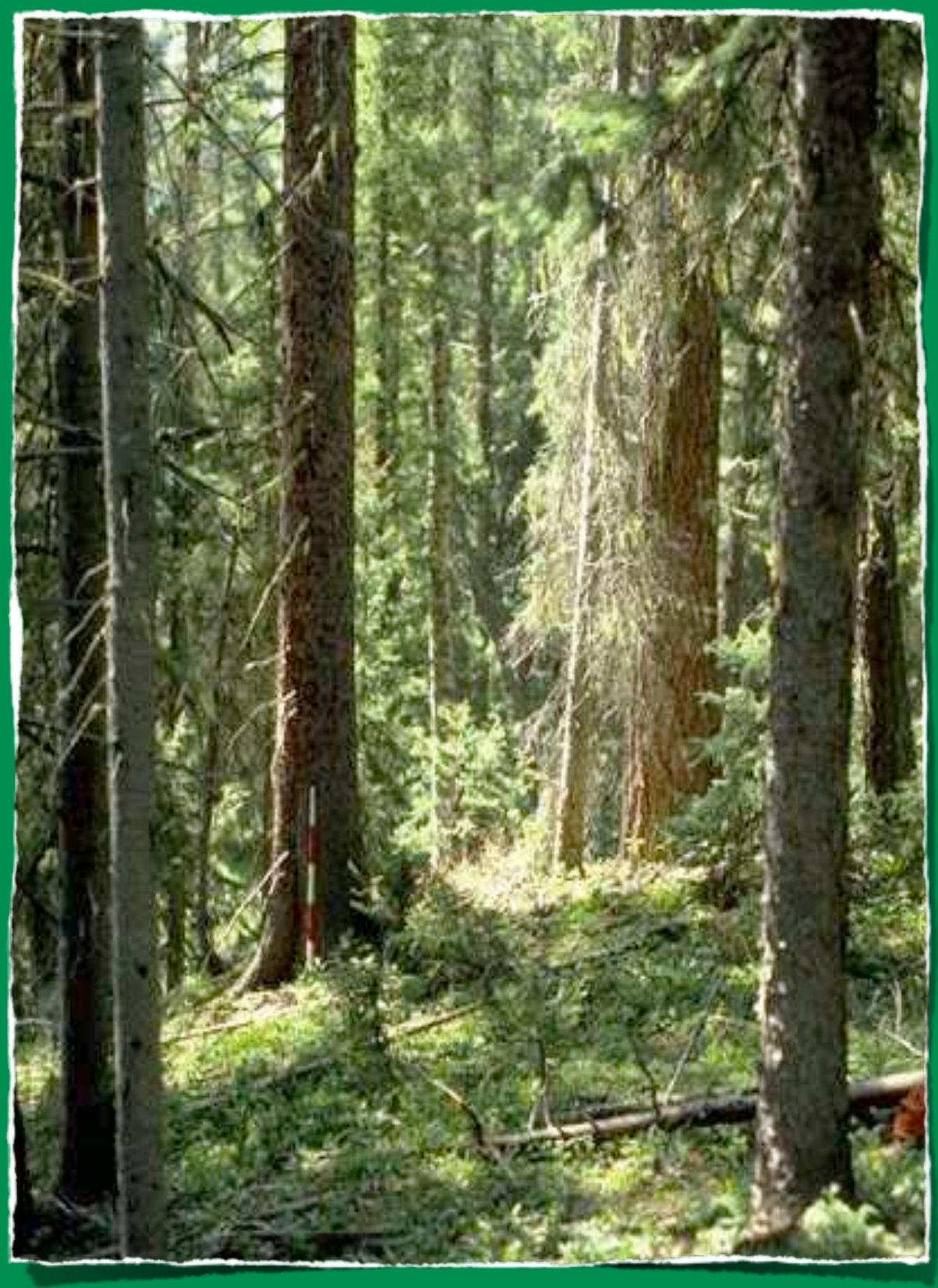


http://www.cwac.net/forests/forest.JPG

Monday, October 26, 2009

You are a programmer on a low-budget B-grade horror movie. Your job is to create the sound of the monster making its way through the forest to the hapless cottage guests.

eliminating redundancy

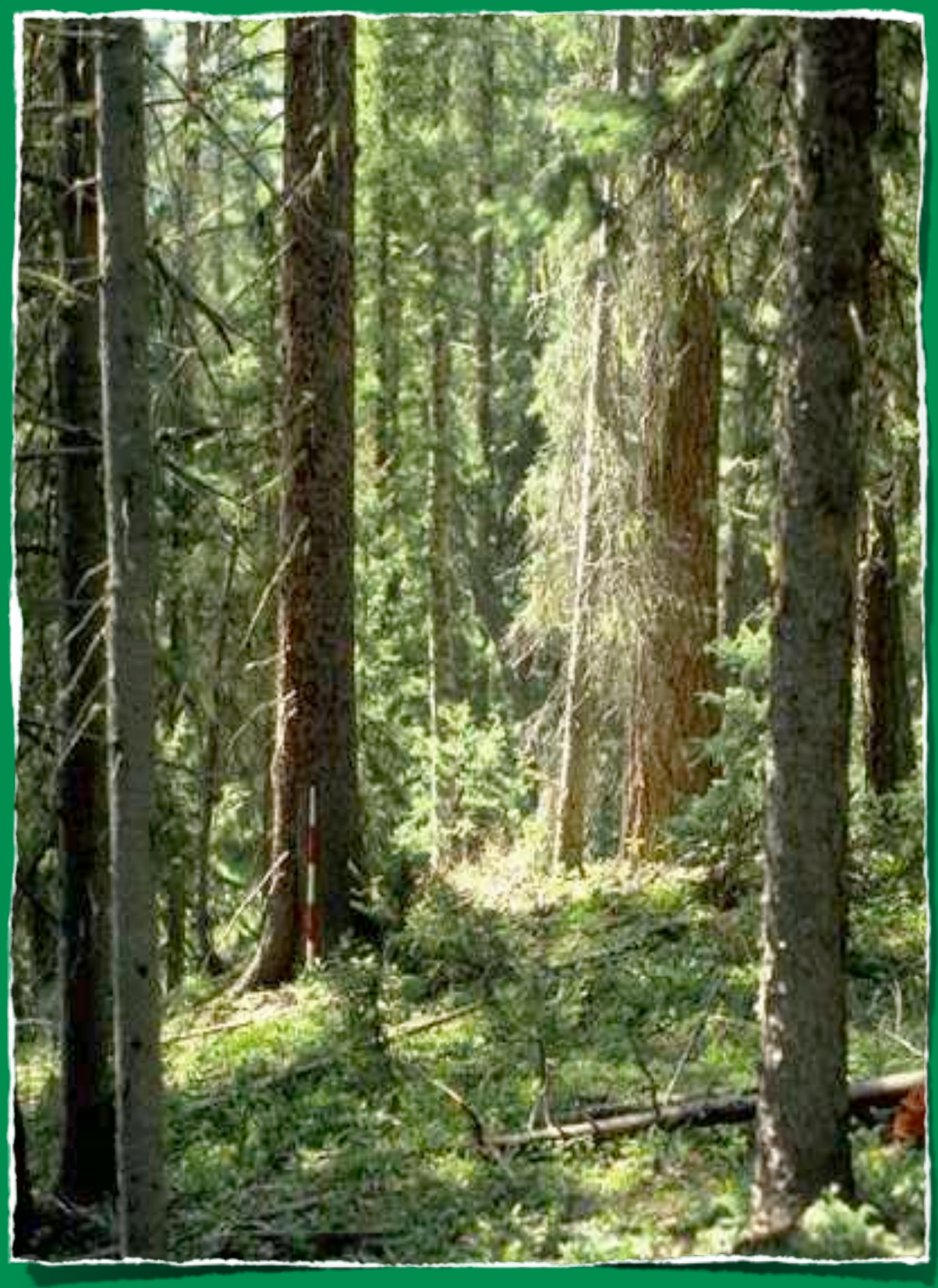


```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

http://www.cwac.net/forests/forest.JPG

You are a programmer on a low-budget B-grade horror movie. Your job is to create the sound of the monster making its way through the forest to the hapless cottage guests.

eliminating redundancy



http://www.cwac.net/forests/forest.JPG

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```


You are a programmer on a low-budget B-grade horror movie. Your job is to create the sound of the monster making its way through the forest to the hapless cottage guests.

eliminating redundancy

http://www.cwac.net/forests/forest.JPG

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```


You are a programmer on a low-budget B-grade horror movie. Your job is to create the sound of the monster making its way through the forest to the hapless cottage guests.

eliminating redundancy

http://www.cwac.net/forests/forest.JPG

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
...
```


You are a programmer on a low-budget B-grade horror movie. Your job is to create the sound of the monster making its way through the forest to the hapless cottage guests.

eliminating redundancy

Instead of typing the guttural utterances over and over, we make a function that does it...

eliminating redundancy

```
def noise():
    print ‘blah blah blargh’
    print ‘blah blah blargh’
    print ‘blargh’
```


Instead of typing the guttural utterances over and over, we make a function that does it...

eliminating redundancy

prints

Monday, October 26, 2009

and call that function three times, which yields the same output.

eliminating redundancy

```
noise()
noise()
noise()
```

prints

and call that function three times, which yields the same output.

eliminating redundancy

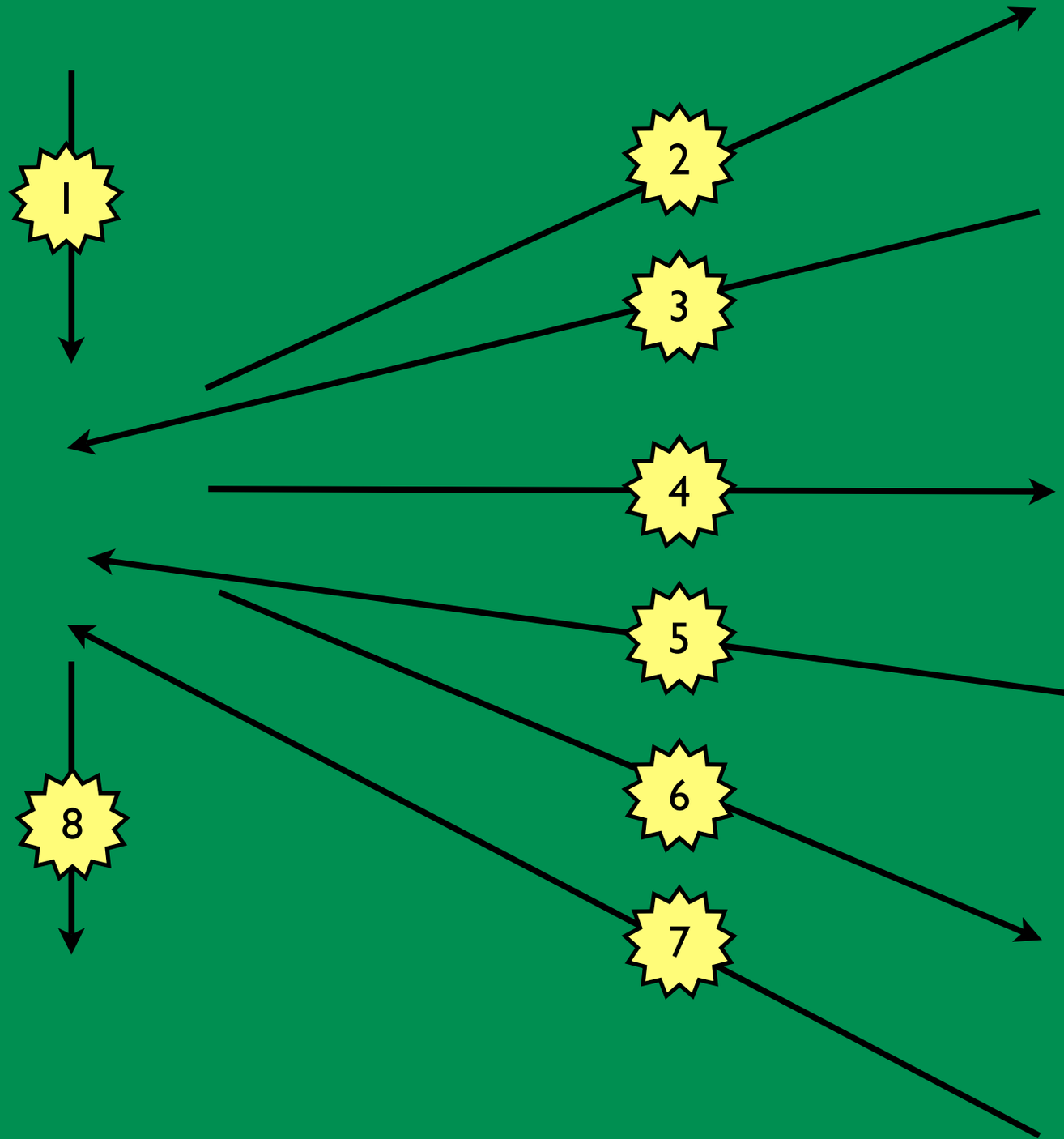
```
noise()
noise()
noise()
```

prints

```
blah blah blargh
blah blah blargh
blargh
blah blah blargh
blah blah blargh
blah blah blargh
blah blah blargh
blargh
```


and call that function three times, which yields the same output.

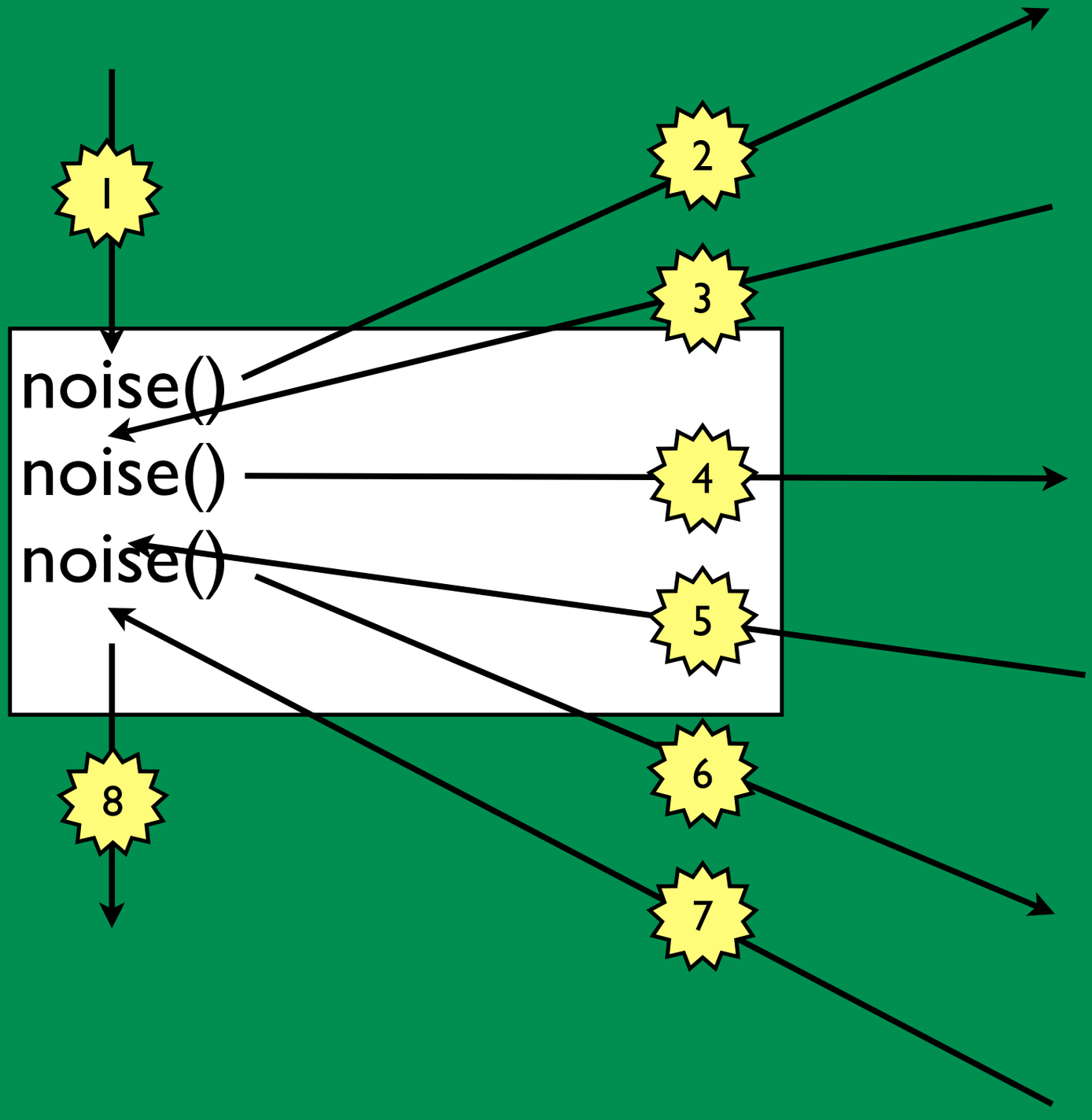
eliminating redundancy



Monday, October 26, 2009

In the main body of the code (left box), each time the noise() function is invoked, execution proceeds into the body of that function until it reaches the end. Then execution returns to where it left off.

eliminating redundancy

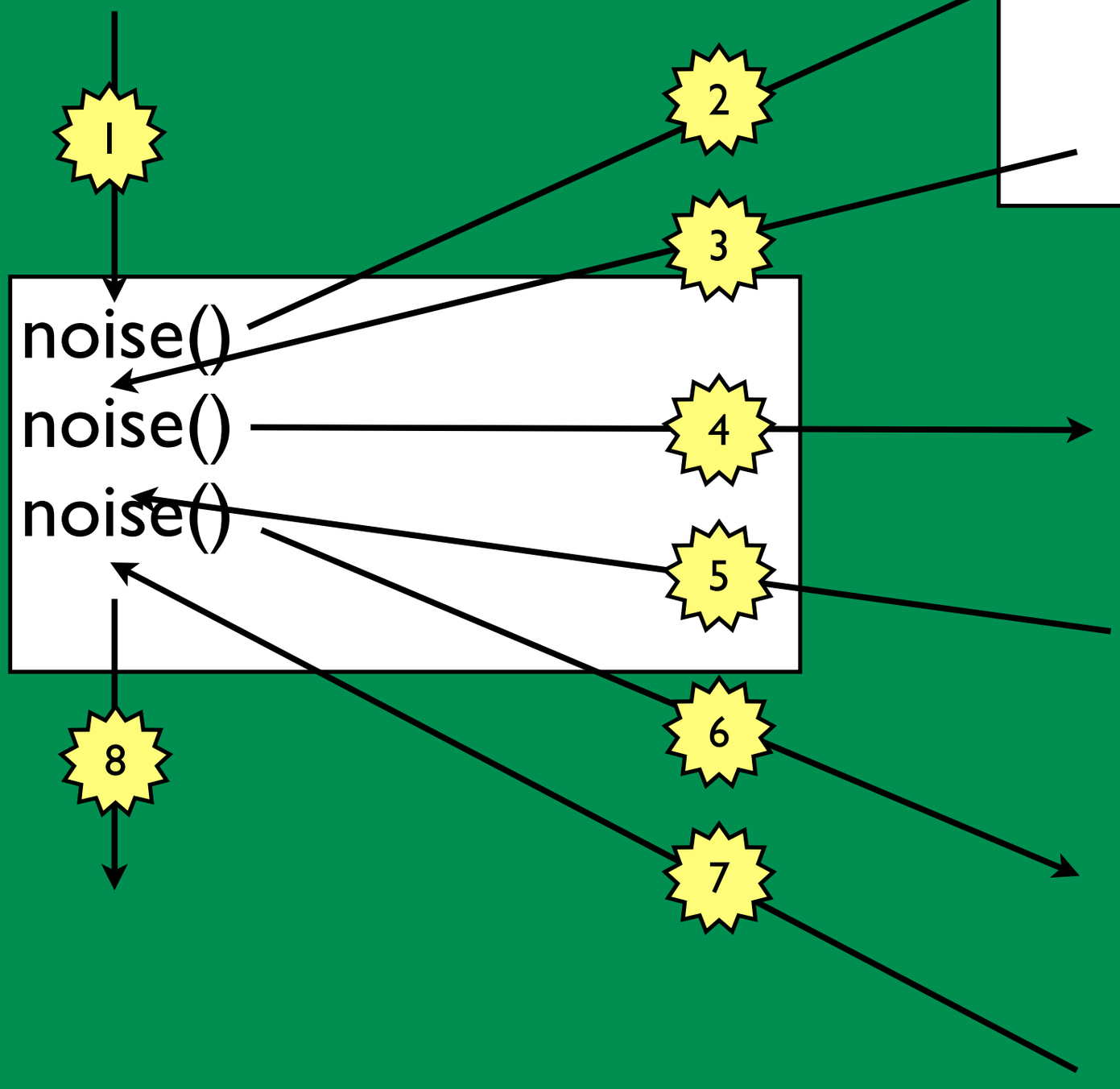


Monday, October 26, 2009

In the main body of the code (left box), each time the noise() function is invoked, execution proceeds into the body of that function until it reaches the end. Then execution returns to where it left off.

eliminating redundancy

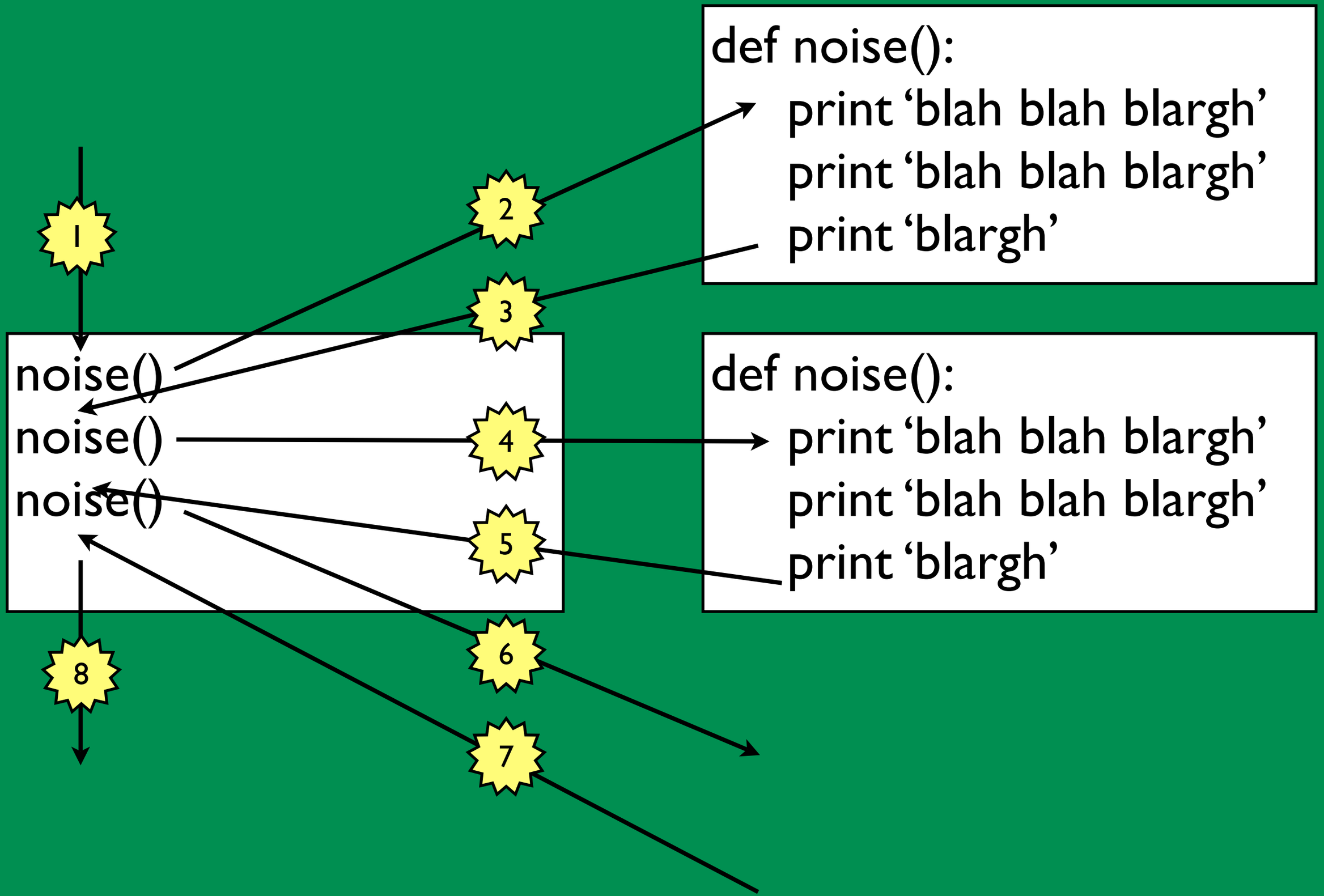
```
def noise():
    print 'blah blah blargh'
    print 'blah blah blargh'
    print 'blargh'
```



Monday, October 26, 2009

In the main body of the code (left box), each time the noise() function is invoked, execution proceeds into the body of that function until it reaches the end. Then execution returns to where it left off.

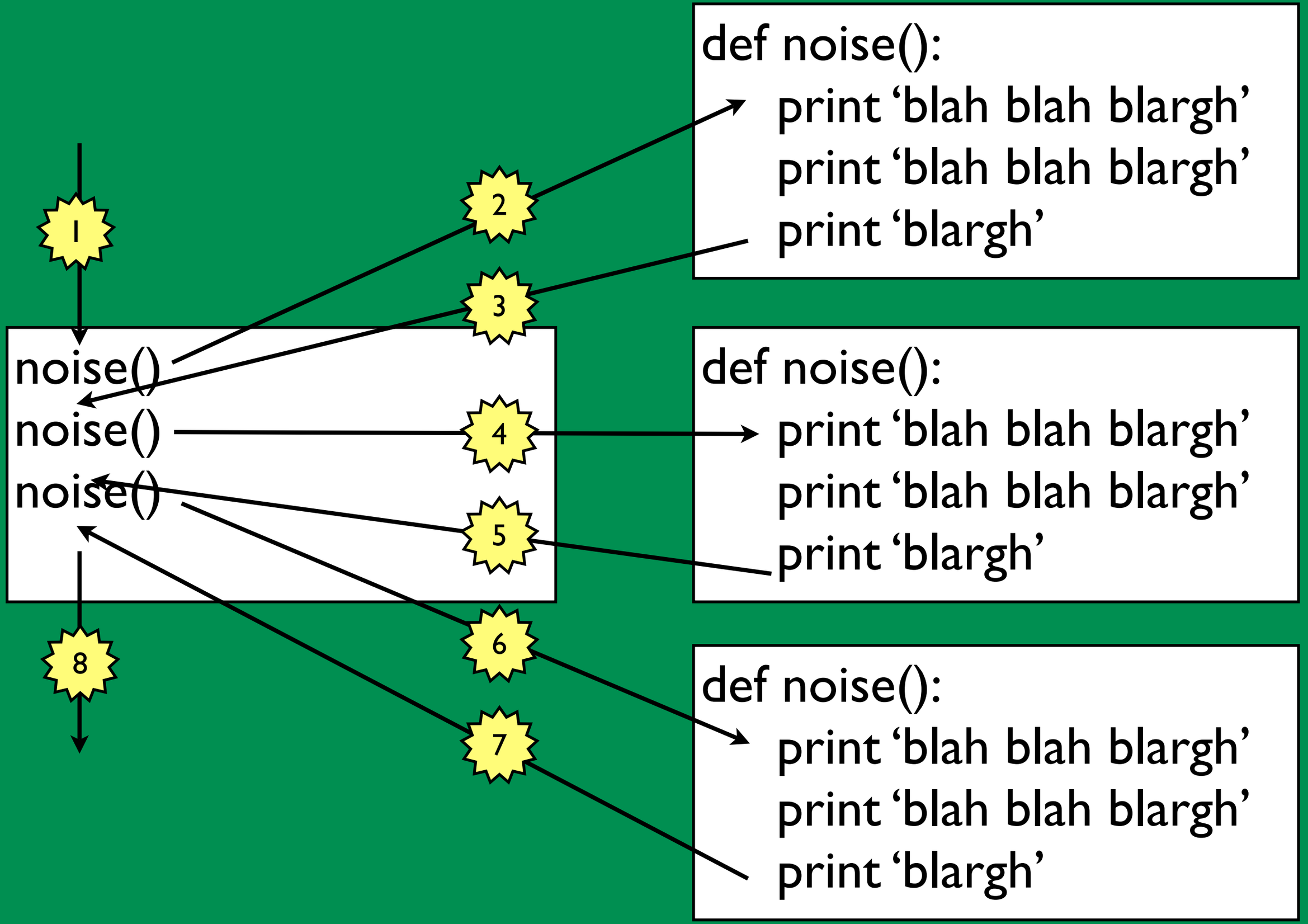
eliminating redundancy



Monday, October 26, 2009

In the main body of the code (left box), each time the noise() function is invoked, execution proceeds into the body of that function until it reaches the end. Then execution returns to where it left off.

eliminating redundancy



In the main body of the code (left box), each time the noise() function is invoked, execution proceeds into the body of that function until it reaches the end. Then execution returns to where it left off.

capitalizing on similarity

http://www.potteryhouse.co.uk/photogallery/spring2006/DSCF7434.JPG
http://tinyurl.com/yzkldcf

Now imagine that we want to vary the monster's noise as it approaches its target. We still have a lot of redundancy, except for the third gurgle in each block.

capitalizing on similarity

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

http://www.potteryhouse.co.uk/photogallery/spring2006/DSCF7434.JPG
http://tinyurl.com/yzkldcf

Now imagine that we want to vary the monster’s noise as it approaches its target. We still have a lot of redundancy, except for the third gurgle in each block.

capitalizing on similarity

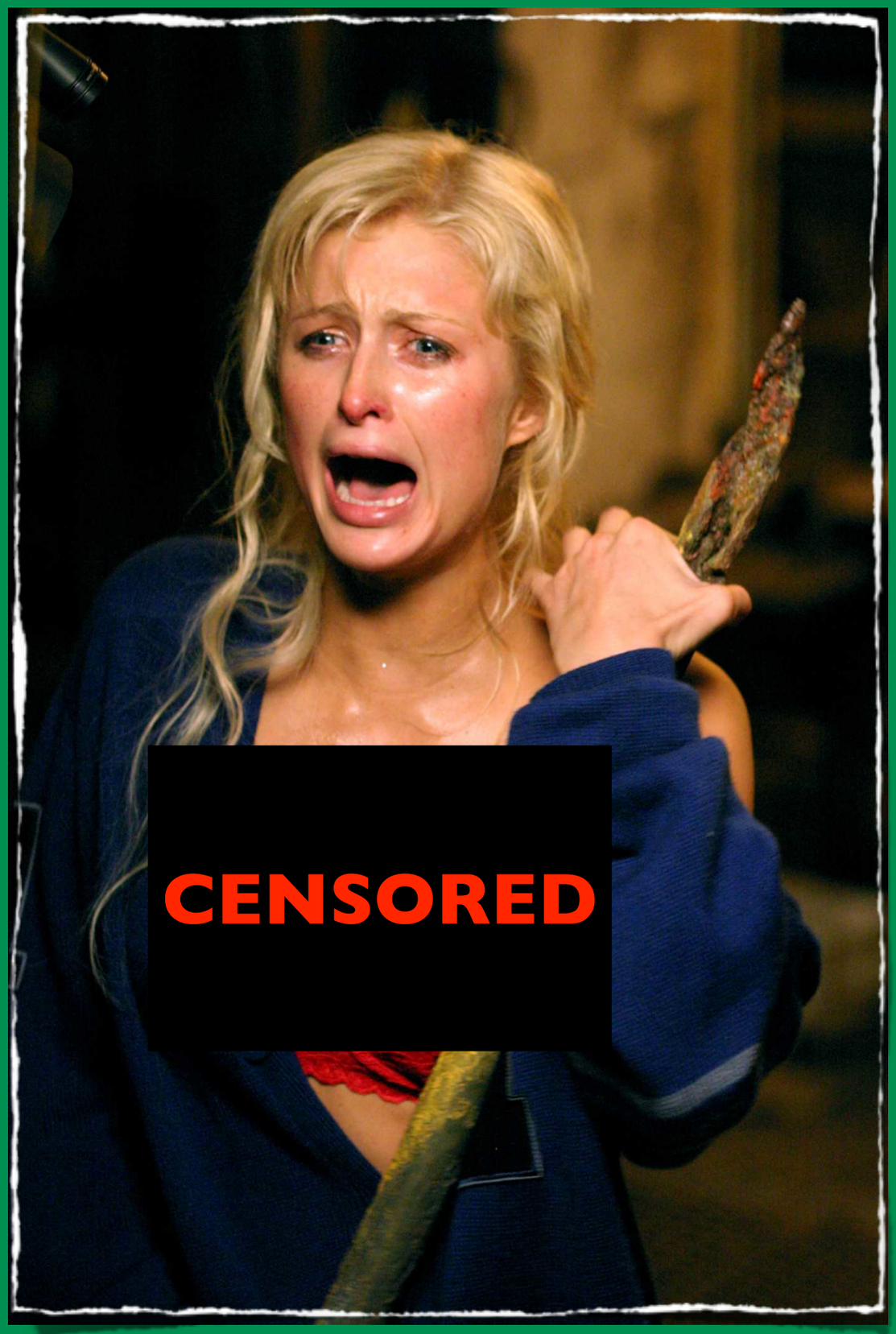
```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘BLAARGH’
```

http://www.potteryhouse.co.uk/photogallery/spring2006/DSCF7434.JPG
http://tinyurl.com/yzkldcf

Now imagine that we want to vary the monster’s noise as it approaches its target. We still have a lot of redundancy, except for the third gurgle in each block.

capitalizing on similarity




```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘blargh’
```

```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘BLAARGH’
```

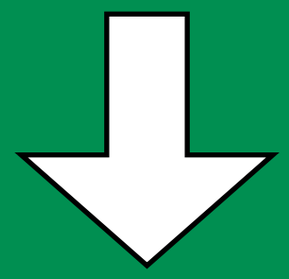
```
print ‘blah blah blargh’
print ‘blah blah blargh’
print ‘BLAAAARGH’
```

http://www.potteryhouse.co.uk/photogallery/spring2006/DSCF7434.JPG
http://tinyurl.com/yzkldcf

Now imagine that we want to vary the monster’s noise as it approaches its target. We still have a lot of redundancy, except for the third gurgle in each block.

Parameterizing the function

```
def noise():
    print 'blah blah blargh'
    print 'blah blah blargh'
    print 'blargh'
```



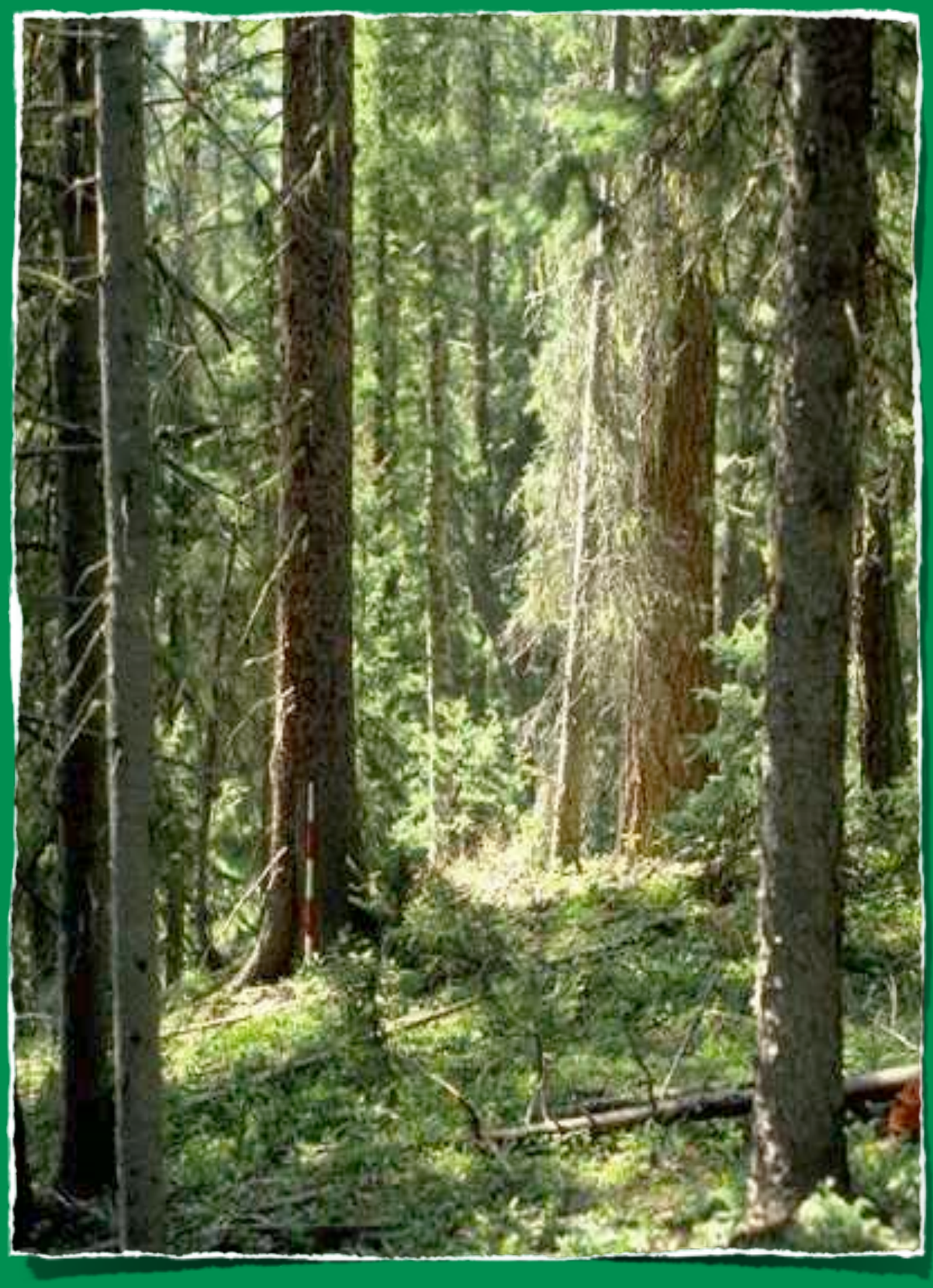
```
def noise(final):
    print 'blah blah blargh'
    print 'blah blah blargh'
    print final
```


By adding a **formal parameter** to the noise() function specification, we allow it to vary what is printed on the third line.

prints

Monday, October 26, 2009

Now we can achieve the same result by passing in a different argument each time we call noise(). An **argument** is what we call the value being passed to the function from the caller's perspective. When we do this, the value that is passed into the function is **bound** to the value of the formal parameter



```
noise('blargh')
```

prints

```
blah blah blargh
blah blah blargh
blargh
```


Now we can achieve the same result by passing in a different argument each time we call noise(). An **argument** is what we call the value being passed to the function from the caller's perspective. When we do this, the value that is passed into the function is **bound** to the value of the formal parameter

```
noise('blargh')
```

```
noise('BLAARGH')
```

prints

```
blah blah blargh
blah blah blargh
blargh
```

```
blah blah blargh
blah blah blargh
BLAARGH
```


Now we can achieve the same result by passing in a different argument each time we call noise(). An **argument** is what we call the value being passed to the function from the caller's perspective. When we do this, the value that is passed into the function is **bound** to the value of the formal parameter


```
noise('blargh')
noise('BLAARGH')
noise('BLAAAARGH')
```

prints

```
blah blah blargh
blah blah blargh
blargh

blah blah blargh
blah blah blargh
BLAARGH

blah blah blargh
blah blah blargh
BLAAAARGH
```


Monday, October 26, 2009

Now we can achieve the same result by passing in a different argument each time we call noise(). An **argument** is what we call the value being passed to the function from the caller's perspective. When we do this, the value that is passed into the function is **bound** to the value of the formal parameter

Using a default value

```
def noise(final):
    print ‘blah blah blargh’
    print ‘blah blah blargh’
    print final
```

```
def noise(final = ‘blargh’):
    print ‘blah blah blargh’
    print ‘blah blah blargh’
    print final
```


Monday, October 26, 2009

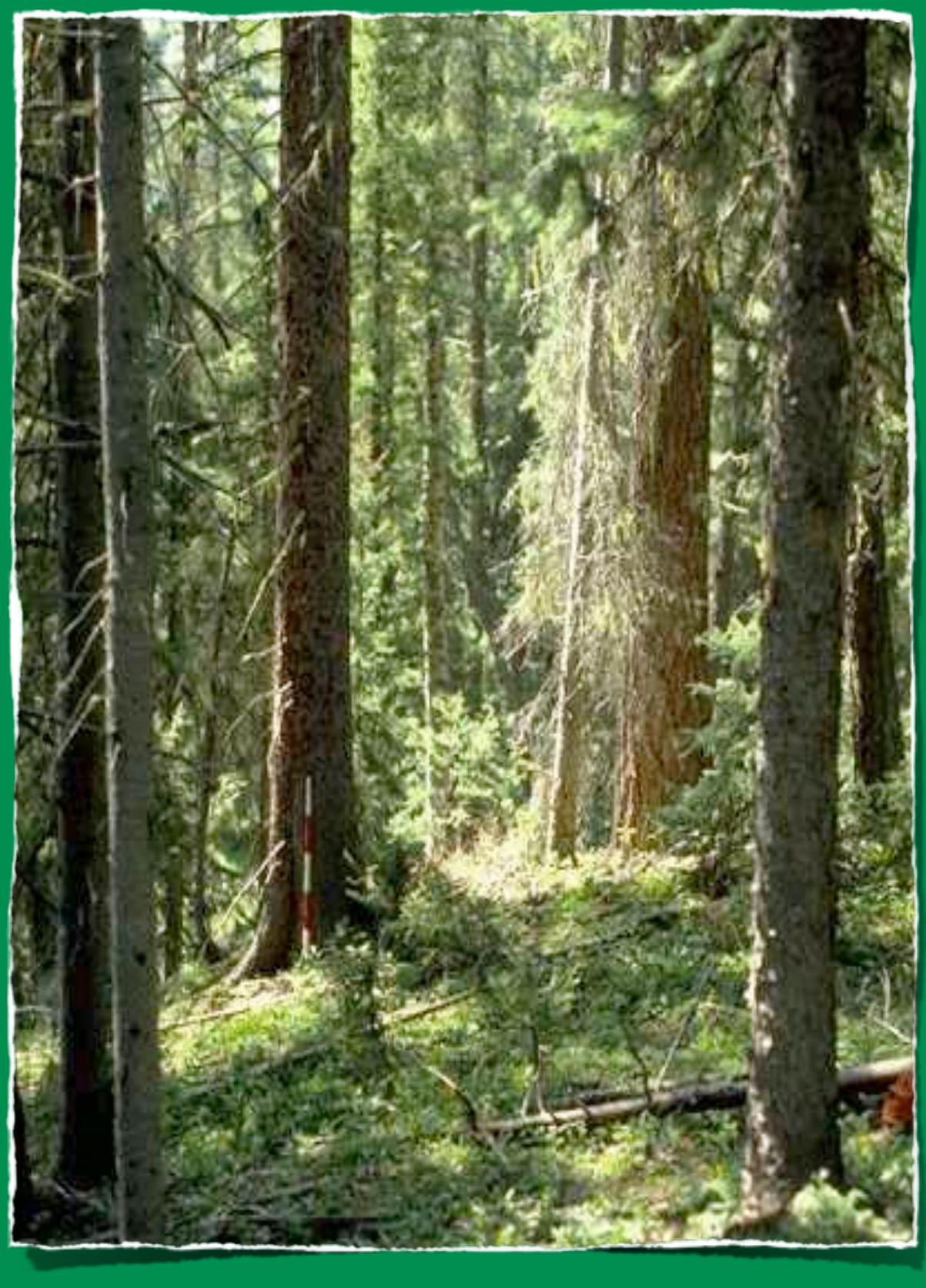
A small note: formal parameters can be given default values, so that if the function is called without any argument, the parameter will receive this default value.

eliminating redundancy

prints (the same)

Monday, October 26, 2009

Here we achieve the same effect by relying on the default value of the noise() function's parameter, "blargh".



```
noise()
```

prints (the same)

```
blah blah blargh
blah blah blargh
blargh
```


Monday, October 26, 2009

Here we achieve the same effect by relying on the default value of the noise() function’s parameter, “blargh”.

eliminating redundancy

```
noise()
```

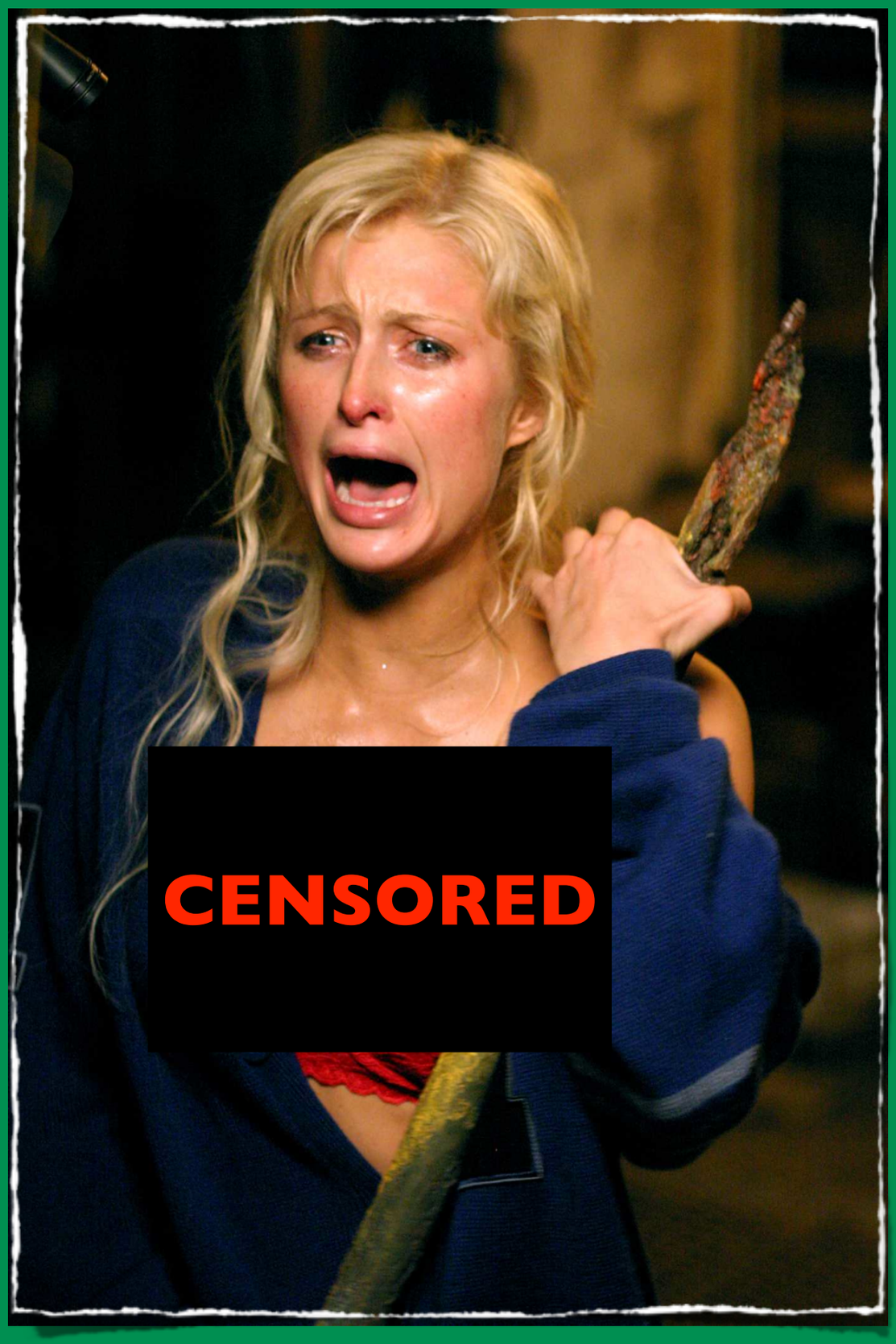
```
noise('BLAARGH')
```

prints (the same)

```
blah blah blargh
blah blah blargh
blargh
```

```
blah blah blargh
blah blah blargh
BLAARGH
```


Here we achieve the same effect by relying on the default value of the noise() function's parameter, "blargh".




```
noise()
noise('BLAARGH')
noise('BLAAAARGH')
```

prints (the same)

```
blah blah blargh
blah blah blargh
blargh
```

```
blah blah blargh
blah blah blargh
BLAARGH
```

```
blah blah blargh
blah blah blargh
BLAAAARGH
```


Here we achieve the same effect by relying on the default value of the noise() function's parameter, "blargh".

RETURN VALUES

```
def noise(final = 'blargh'):
    print 'blah blah blargh'
    print 'blah blah blargh'
    print final
```

```
def noise(final = 'blargh'):
    print 'blah blah blargh'
    print 'blah blah blargh'
    print final
    return len(final)
```


Monday, October 26, 2009

Every function returns a value to the caller when it is finished executing. If you don't specify a return value (using the **return** statement), the default value of None is used. Here, we return the length of the final monster noise as a gross approximation of how scary the noise is (we are admittedly

return values

prints

Monday, October 26, 2009

Now that noise() returns a value, we capture it by assigning it to a variable and printing it out.

```
noise()
noise('BLAARGH')
scary_factor =
    noise('BLAAAARGH')


print scary_factor
```

prints

Now that noise() returns a value, we capture it by assigning it to a variable and printing it out.

return values

```
noise()
noise('BLAARGH')
scary_factor =
    noise('BLAAAARGH')

print scary_factor
```

prints

```
9
```


Now that noise() returns a value, we capture it by assigning it to a variable and printing it out.

Return values

As we noted before, previously the noise() function did not return any values, so the return value defaulted to None. This was okay, since we were not using the return value, but in general it's a good idea to explicitly list the return value you'd like to use, even if you don't plan on using it, to avoid

Return values

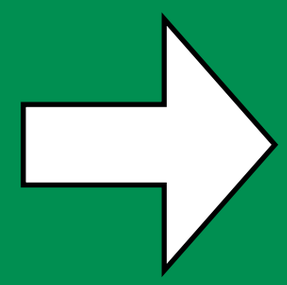
```
def noise(final = 'blargh'):
    print 'blah blah blargh'
    print 'blah blah blargh'
    print final
    return len(final)
```


As we noted before, previously the noise() function did not return any values, so the return value defaulted to None. This was okay, since we were not using the return value, but in general it's a good idea to explicitly list the return value you'd like to use, even if you don't plan on using it, to avoid

Return values

```
def noise(final = 'blargh'):
    print 'blah blah blargh'
    print 'blah blah blargh'
    print final
    return len(final)
```

```
def noise(final = 'blargh'):
    print 'blah blah blargh'
    print 'blah blah blargh'
    print final
```



returns None

As we noted before, previously the noise() function did not return any values, so the return value defaulted to None. This was okay, since we were not using the return value, but in general it's a good idea to explicitly list the return value you'd like to use, even if you don't plan on using it, to avoid

Why?

READABILITY (MODULARITY)

Functions also improve the readability of code, which helps with long-term (and even short-term) maintainability of the code. This is a real issue with huge software projects (like, say, Microsoft Office) that have tens of millions of lines of code, but is also an issue with smaller projects. It can be hard to

What does this code do?

The square() function helps you remember what you meant by x * x (since square() has a quick human interpretation). That's not that useful here, where an experienced programmer can read both of them in equal time.

What does this code do?

```
# x is an integer
value = x * x
```


The square() function helps you remember what you meant by x * x (since square() has a quick human interpretation). That's not that useful here, where an experienced programmer can read both of them in equal time.

What does this code do?

```
# x is an integer
value = x * x
```

```
# x is an integer
value = square(x)
```


The square() function helps you remember what you meant by x * x (since square() has a quick human interpretation). That's not that useful here, where an experienced programmer can read both of them in equal time.

What does this code do?

It's more of an issue here. It might take you a minute to recognize that the code in the first box computes the length of a hypotenuse of a right triangle (the Euclidean distance between two points on a plane). If you put that code in a function instead, the meaning would be immediate.

What does this code do?

```
# p1 and p2 are Point objects
value = math.sqrt(square(p2.getX() - p1.getX())
                  + square(p2.getY() - p1.getY())
```


It's more of an issue here. It might take you a minute to recognize that the code in the first box computes the length of a hypotenuse of a right triangle (the Euclidean distance between two points on a plane). If you put that code in a function instead, the meaning would be immediate.

What does this code do?

```
# p1 and p2 are Point objects
value = math.sqrt(square(p2.getX() - p1.getX())
                  + square(p2.getY() - p1.getY())
```

```
# p1 and p2 are Point objects
value = Euclidean_distance(p1,p2)
```


It's more of an issue here. It might take you a minute to recognize that the code in the first box computes the length of a hypotenuse of a right triangle (the Euclidean distance between two points on a plane). If you put that code in a function instead, the meaning would be immediate.

What does this code do?

Even if you don’t know what the prime factorization of an integer is, you know what’s going on in Box 2, which wouldn’t be the case for Box 1.

What does this code do?

```
# x is an integer
list = []
for i in range(2,x+1):
        while x % i == 0:
            list.append(i)
            x /= i
return list
```


Even if you don't know what the prime factorization of an integer is, you know what's going on in Box 2, which wouldn't be the case for Box 1.

What does this code do?

```
# x is an integer
list = []
for i in range(2,x+1):
        while x % i == 0:
            list.append(i)
            x /= i
return list
```

```
# x is an integer
factors = prime_factors(x)
```


Even if you don't know what the prime factorization of an integer is, you know what's going on in Box 2, which wouldn't be the case for Box 1.

Topic:

VARIABLES AND SCOPE

Anatomy of a Function

```
def myFunc(param1,param2):
    var1 = var2 = 1
    #
    # do some stuff
    #
    return var1,var2
```


A function definition tells you how a function will work when it is called. A function has a name and a list of comma-separated parameters (these parameters are bound to the **actual parameters** when the function is called). You can have as many parameters as you like, and as few as zero.

Anatomy of a Function

information in
(as many args as you want)

```
def myFunc(param1,param2):
    var1 = var2 = 1
    #
    # do some stuff
    #
    return var1,var2
```


A function definition tells you how a function will work when it is called. A function has a name and a list of comma-separated parameters (these parameters are bound to the **actual parameters** when the function is called). You can have as many parameters as you like, and as few as zero.

Anatomy of a Function

```
def myFunc(param1,param2):
    var1 = var2 = 1
    #
    # do some stuff
    #
    return var1, var2
```


A function also has internal variables. **These variables are not visible outside the function.** The only way for a function to communicate with its caller is with its return values.

Anatomy of a Function

information in
(as many args as you want)

variables
(visible only in the function)

```
def myFunc(param1,param2):
    var1 = var2 = 1
    #
    # do some stuff
    #
    return var1, var2
```


A function also has internal variables. **These variables are not visible outside the function.** The only way for a function to communicate with its caller is with its return values.

Anatomy of a Function

```
def myFunc(param1,param2):
    var1 = var2 = 1
    #
    # do some stuff
    #
    return var1,var2
```


You can return as many variables as you'd like. On the calling side, you assign them like this:

```
x,y = myFunc(arg1,arg2)
```

Anatomy of a Function

```
def myFunc(param1,param2):
    var1 = var2 = 1
    #
    # do some stuff
    #
    return var1,var2
```

information in
(as many args as you want)

variables
(visible only in the function)

information out
(as many return values as you want)

Monday, October 26, 2009

You can return as many variables as you'd like. On the calling side, you assign them like this:

```
x,y = myFunc(arg1,arg2)
```

Calling a Function

```
def square(n):
    n2 = n * n
    return n2
```

```
def main():
    x = input("value? ")
    print x, "squared is", square(x)
```


Monday, October 26, 2009

Here we see the difference between the function definition (for square()) and it being called. When main calls square(x), the value of x in main() is assigned to the formal parameter n in square().

Calling a Function

```
def square(n):
    n2 = n * n
    return n2
```

n is a **formal parameter** (part of the function definition)

```
def main():
    x = input("value? ")
    print x, "squared is", square(x)
```


Monday, October 26, 2009

Here we see the difference between the function definition (for square()) and it being called. When main calls square(x), the value of x in main() is assigned to the formal parameter n in square().

Calling a Function

```
def square(n):
    n2 = n * n
    return n2
```

n is a **formal parameter** (part of the function definition)

```
def main():
    x = input("value? ")
    print x, "squared is", square(x)
```

when square(*x*) is called, the value of *x* (the **actual parameter** is assigned to *n*)

Monday, October 26, 2009

Here we see the difference between the function definition (for square()) and it being called. When main calls square(x), the value of x in main() is assigned to the formal parameter n in square().

Some Important Notes

Some Important Notes

- function arguments are passed *by value*
  - this means that the *value* of the variable is passed, not the variable itself
  - so if the function changes the value of the variable, the caller won't see the changes
  - the caller and the callee may both have different names for the value; neither knows the names the other uses (or cares)

Summary

Summary

- **Scope**
  - All variables used by a function must be (a) declared inside the function or (b) passed in as arguments
  - All variables declared in a function cease to exist afterward

Summary

- **Scope**
  - All variables used by a function must be (a) declared inside the function or (b) passed in as arguments
  - All variables declared in a function cease to exist afterward
- **Pass by value**
  - Any modifications to arguments are lost

Summary

- **Scope**
  - All variables used by a function must be (a) declared inside the function or (b) passed in as arguments
  - All variables declared in a function cease to exist afterward
- **Pass by value**
  - Any modifications to arguments are lost
- **Communication**
  - In (via passed arguments) and out (via return value(s))

EXAMPLES

See annotated python examples (separate).