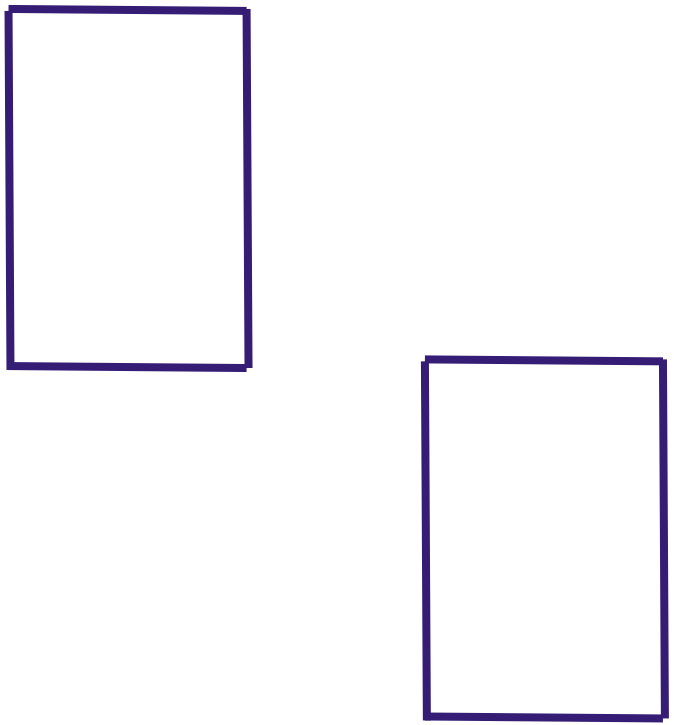
FN.1

Concept of Functions as Named Blocks of Code

Functions

Organizing Code by
Naming Groups of Instructions

Organizing Instructions



Organizing Instructions

```
Move forward by 100 steps
```

Organizing Instructions

```
Move forward by 100 steps
Turn left
```

Organizing Instructions

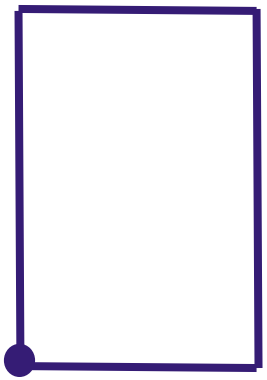
```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
```

Organizing Instructions

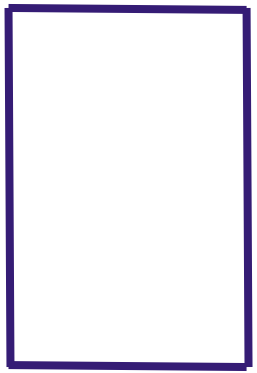
```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```



Organizing Instructions

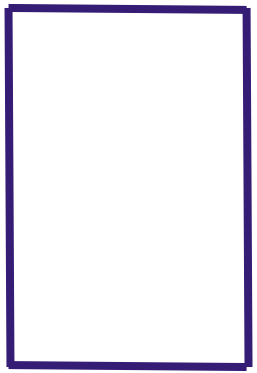
```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left


Pick up pen
Move to new location
Put down pen
```



Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

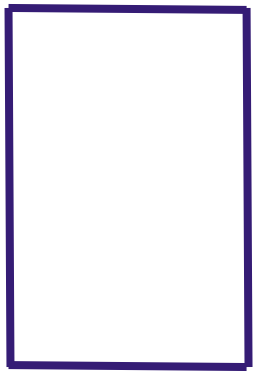


```
Pick up pen
Move to new location
Put down pen
```

```
Move forward by 100 steps
Turn left
```

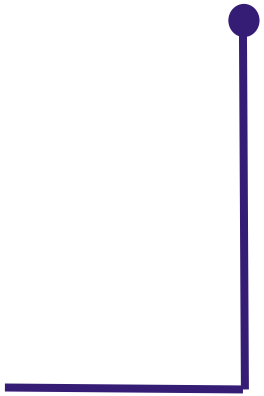
Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```



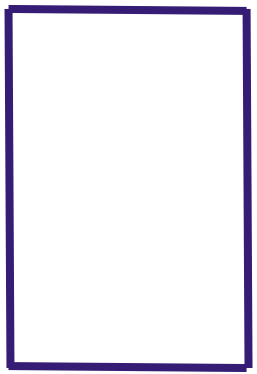
```
Pick up pen
Move to new location
Put down pen
```

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```



Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

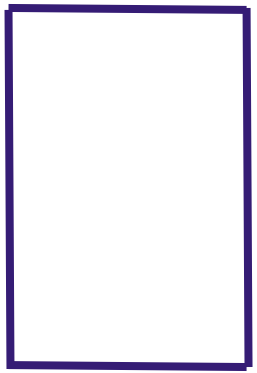


```
Pick up pen
Move to new location
Put down pen
```

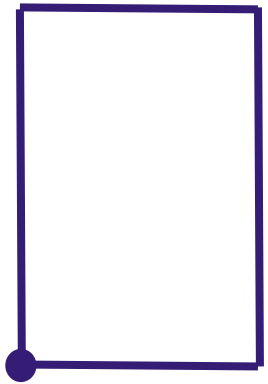
```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
```

Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```



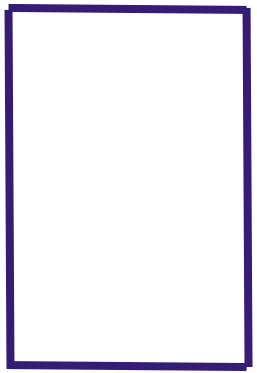
```
Pick up pen
Move to new location
Put down pen
```



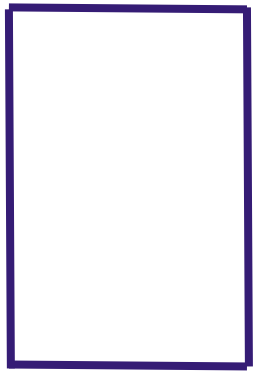
```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```



```
Pick up pen
Move to new location
Put down pen
```



```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

```
Pick up pen
Move to new location
Put down pen
```

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

Problem

Repeated Blocks of Identical Instructions

- Makes code longer (more to read, harder to absorb overall point)
- Doesn’t explicitly demonstrate organization.

Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

```
Pick up pen
Move to new location
Put down pen
```

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

Problem

Repeated Blocks of Identical Instructions

- Makes code longer (more to read, harder to absorb overall point)
- Doesn't explicitly demonstrate organization.

Solution

- Group the instructions together explicitly and give them a single name.
- Run those instructions using the name, rather than copying the code and running it.

Organizing Instructions

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

```
Pick up pen
Move to new location
Put down pen
```

```
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
Move forward by 100 steps
Turn left
Move forward by 200 steps
Turn left
```

Function: named block of instructions with a particular purpose

```
function draw_rectangle is
begin
     Move forward by 100 steps
     Turn left
     Move forward by 200 steps
     Turn left
     Move forward by 100 steps
     Turn left
     Move forward by 200 steps
     Turn left
end
```

Organizing Instructions

Function: named block of instructions with a particular purpose

```
draw_rectangle()

Pick up pen
Move to new location
Put down pen

draw_rectangle()
```

```
function draw_rectangle is
begin
     Move forward by 100 steps
     Turn left
     Move forward by 200 steps
     Turn left
     Move forward by 100 steps
     Turn left
     Move forward by 200 steps
     Turn left
end
```

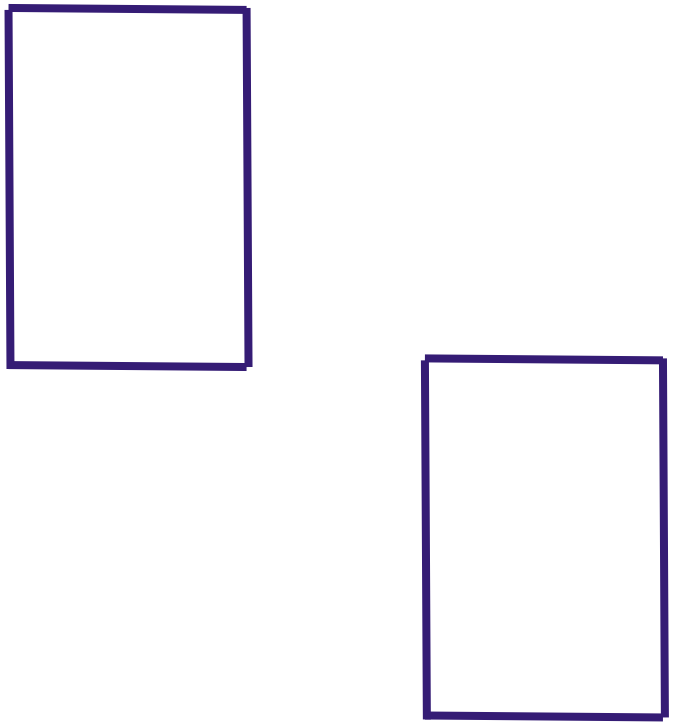
Organized Instructions

```
function draw_rectangle is
begin
     Move forward by 100 steps
     Turn left
     Move forward by 200 steps
     Turn left
     Move forward by 100 steps
     Turn left
     Move forward by 200 steps
     Turn left
end


draw_rectangle()

Pick up pen
Move to new location
Put down pen

draw_rectangle()
```



Functions are Everywhere!

Built into languages

- Print statements
- Functions that read in data files

As part of libraries

- Functions implementing machine learning methods to analyze data
- Functions implementing helpful plots

In the code that we write to effectively use languages and libraries

FN.2

Concept of Calling Built-in Functions

Calling Built-in Functions

Calling Functions

To take advantage of functions that are part of the programming language, we need to **call** them.

```
print(“Hello, World”)
```

Calling Functions

To take advantage of functions that are part of the programming language, we need to **call** them.

function name

```
print(“Hello, World”)
```

Calling Functions

To take advantage of functions that are part of the programming language, we need to **call** them.

function name function input

```
print(“Hello, World”)
```

Calling Functions

To take advantage of functions that are part of the programming language, we need to **call** them.

```
print(“Hello, World”)
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes

Calling Functions

To take advantage of functions that are part of the programming language, we need to **call** them.

Printed to screen

```
print("Hello, World")
```

```
Hello, World
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes

Calling Functions that Return Values

Some functions return values.

```
compute_average(
```

5	6	7

```
)
```

Calling Functions that Return Values

Some functions return values.

```
compute_average( 5 6 7 )
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values.

```
compute_average( 5 | 6 | 7 )
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values.

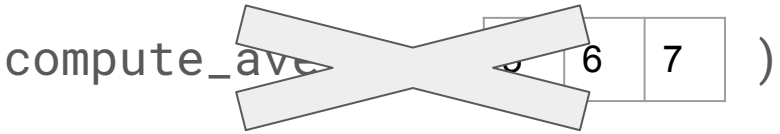
```
compute_average( 5 | 6 | 7 )
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values.



Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values.

6

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it.

```
mean <- compute_average(5, 6, 7)
```

First: Call/Execute/Invoke

Second: Assign value to a variable

Name	Value

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it.

```
mean <- compute_average(5, 6, 7)
```

Name	Value

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it.

```
mean <- compute_average(5, 6, 7)
```

Name	Value

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it.

```
mean <- compute_a( nums )
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Name	Value

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it.

Name	Value

```
mean <- 6
```

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it.

```
mean <- 6
```

Finish Assignment

Name	Value
`mean`	6

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it. Input must be a value, so if it is a variable, we need to look up the value.

```
mean <- compute_average( nums )
```

Name	Value		
nums	5	6	7

First: Call/Execute/Invoke

Second: Assign value to a variable

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it. Input must be a value, so if it is a variable, we need to look up the value.

```
mean <- compute_average(5, 6, 7)
```

Name	Value
nums	5 6 7

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it. Input must be a value, so if it is a variable, we need to look up the value.

```
mean <- compute_average(5, 6, 7)
```

Name	Value
nums	5 6 7

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it. Input must be a value, so if it is a variable, we need to look up the value.

```
mean <- compute_a( nums )
```

Name	Value
nums	5 6 7

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it. Input must be a value, so if it is a variable, we need to look up the value.

```
mean <- 6
```

Name	Value		
nums	5	6	7

Call/Execute/Invoke

Step 1: Value of input is Sent to Function
Step 2: Function Executes
Step 3: Value is Returned from Function

Calling Functions that Return Values

Some functions return values. In the code that **calls** that function, we need to capture that return value and do something with it. Input must be a value, so if it is a variable, we need to look up the value.

```
mean <- 6
```

Finish Assignment

Name	Value
`nums`	5 6 7
`mean`	6

V.FN.3R

Calling functions in R

Outline of plan

- Use RStudio console
- Start with print to introduce the term argument.
- Use seq to create different sequences
  - Show help for seq to show parameters and their default values. seq()
  - Introduce concept of multiple arguments. seq(1,5)
  - Show matching by position and matching by name seq(from=1,to=5), seq(to=5, from=1), seq(1,to=5,by=2), seq(1,2,to=5)
  - Then show that arguments can be literal values, values of variables, or the result of an expression
- Use mean to find average of [5,6,7]
  - Show that we can put the output of one function as the input of another

V.FN.3P

Calling functions in Python

Outline of plan

- Present each slide as a slide, then show code in “visualize execution mode” in PythonTutor

V.FN.3P

Calling functions in Python

Calling a Python function with an Argument

- Each input is referred to as an **argument**
- An argument can be
  - A literal value (e.g. 4, or “hello”)

```
num1 = 42

num2 = 31

print( "hello" )
```

Argument

Calling a Python function with an Argument

- Each input is referred to as an **argument**
- An argument can be
  - A literal value (e.g. 4, or “hello”)
  - The value of a variable

```
num1 = 42

num2 = 31

print( "hello" )

print( num1 )
```

Argument

Calling a Python function with an Argument

- Each input is referred to as an **argument**
- An argument can be
  - A literal value (e.g. 4, or “hello”)
  - The value of a variable
  - Something more complicated, such as result of an expression

```
num1 = 42

num2 = 31

print( "hello" )

print( num1 )

print( num1+num2 )
```

Argument

Calling a Python function with Positional Arguments

- When there is more than one argument, how does Python know what to print first?
  - It examines the arguments by position.
  - So, a more precise term for each arguments in this example is **positional argument**

```
num1 = 42

num2 = 31

print( "hello" )

print( "Nums are", num1, num2 )

print( "Their sum is", num1+num2 )
```

Positional Argument

Positional Argument

Calling a Python function with Keyword Arguments

- When we supply the parameter's name, it is called a **keyword argument**
- This allows us to write more readable code. We know what the purpose of each argument is
- Rule: Keyword arguments must come after all positional arguments

```
num1 = 42

num2 = 31

print( num1, num2, sep="+")

print( sep="+", num1 )
```

Keyword Argument

Illegal: positional argument after keyword argument

Calling a Python function with output from another function

```
nums = [5,6,7]

N = len(nums)

print( "Printing len from variable", N )

print( "Printing len from function output", len( nums ) )

mean = sum(nums) / len(nums)

print( "Average is", mean )
```

Calls **len** and then passes result to **print**