Functions

Reusing Code

Reusing Code

Sometimes, you want to run the same block of commands and multiple different locations in your code.

Reusing Code

Sometimes, you want to run the same block of commands and multiple different locations in your code.

Can't just use a for-loop since these locations may be separated by a region of code that you only want to execute once.

Reusing Code

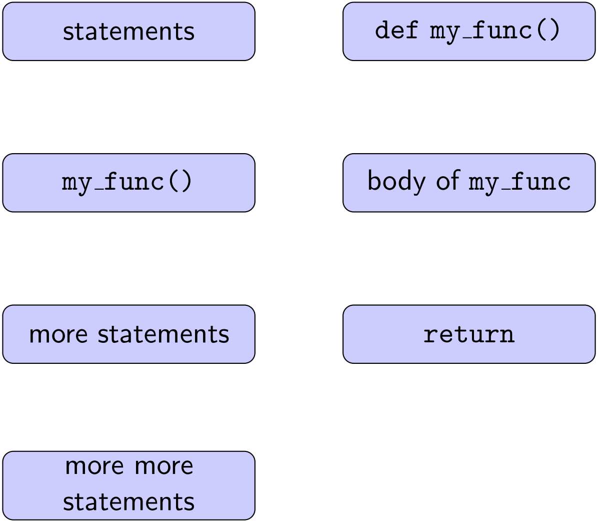
```
print("Instructions:")
print("Pick a digit between 1 and 10")
print("Don’t tell your opponent what you picked")
player1 = int(input("Player 1:  enter pick here:  "))
print("Instructions:")
print("Pick a digit between 1 and 10")
print("Don’t tell your opponent what you picked")
player2 = int(input("Player 2:  enter pick here:  "))
if player2 == player1:
    print("Player 2 wins!!!")
else:
    print("Player 1 wins!!!")
```

Reusing Code

```
def instructions():
    print("Instructions:")
    print("Pick a digit between 1 and 10")
    print("Don’t tell your opponent what you picked")

instructions()
player1 = int(input("Player 1:  enter pick here:  "))
instructions()
player2 = int(input("Player 2:  enter pick here:  "))
if player2 == player1:
    print("Player 2 wins!!!")
else:
    print("Player 1 wins!!!")
```

Function Control Flow



statements

my_func()

more statements

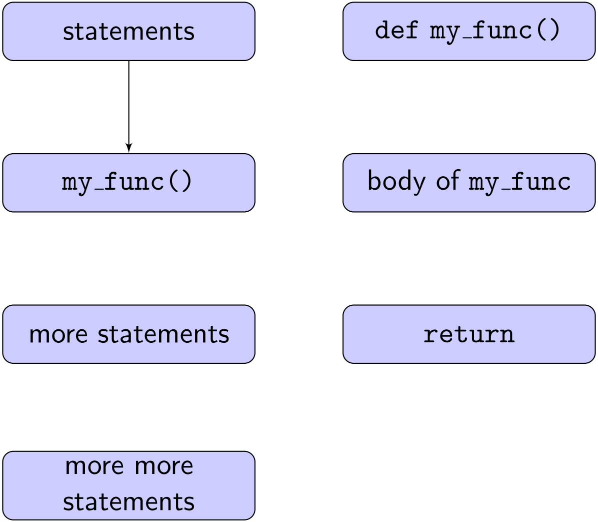
more more statements

def my_func()

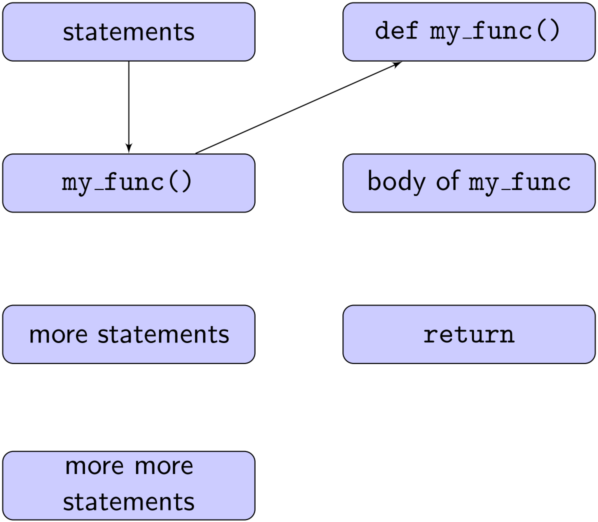
body of my_func

return

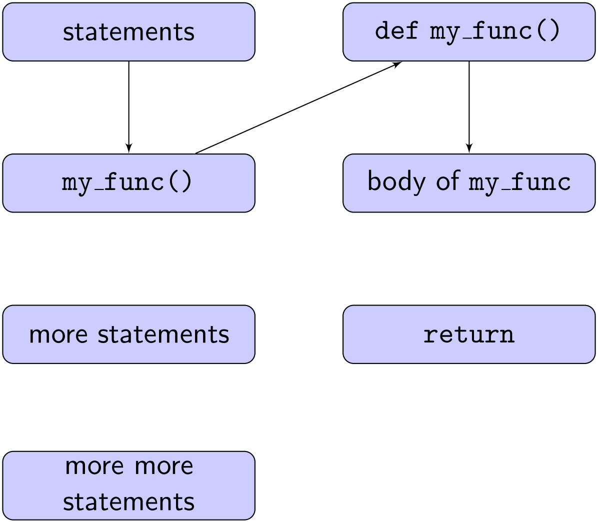
Function Control Flow



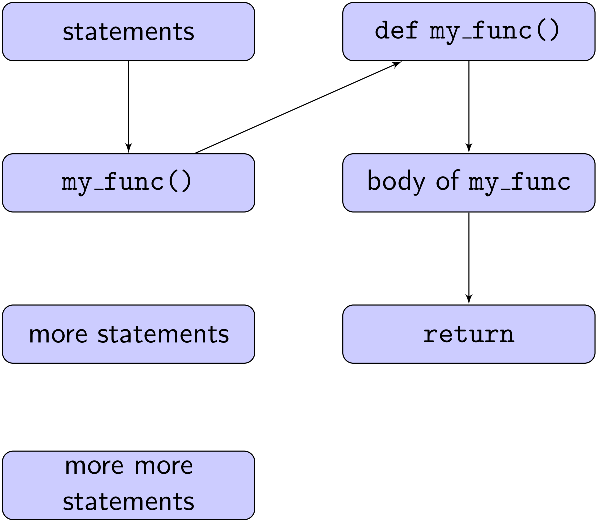
Function Control Flow



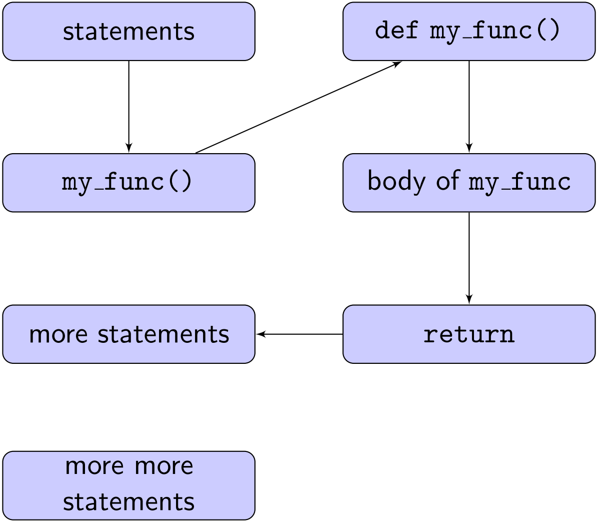
Function Control Flow



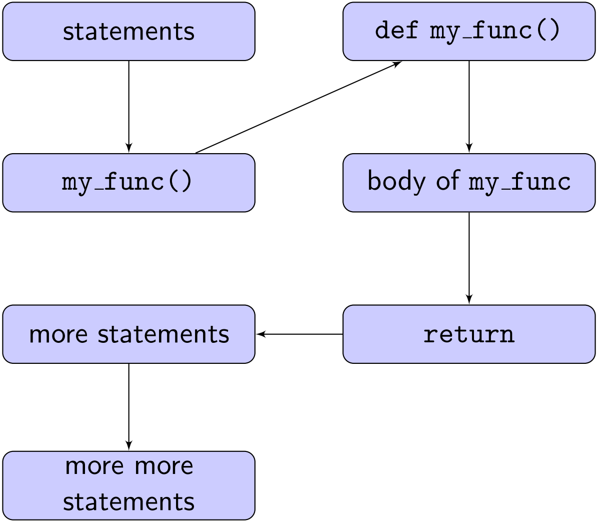
Function Control Flow



Function Control Flow



Function Control Flow



Garbage Collector

Actual footage of a real-life variable being released:

```
>>> x = 1
>>> x = 2
>>> x = x + 1
>>> _
```

Memory:

Address	Contents
00000000	01101011
00000001	11001100
x	00000011
00000011	10101110
⋮	⋮
11111111	00100000

Garbage Collector

Actual footage of a real-life variable being released:

```
>>> x = 1
>>> x = 2
>>> x = x + 1
>>> quit()
```

Memory:

Address	Contents
00000000	01101011
00000001	11001100
00000010	00000011
00000011	10101110
⋮	⋮
11111111	00100000

Garbage Collector

Actual footage of a funcion return:

```
def my_func():
    x = 2
    x = x + 1
    return
```

Memory:

Address	Contents
00000000	01101011
00000001	11001100
00000010	00000011
00000011	10101110
⋮	⋮
11111111	00100000

Concept Check!

What is printed by each of the following?

1.
```
x = 0
def my_func():
    x = 1
    print(x)
    return

my_func()
```

2.
```
x = 0
def my_func():
    x = 1
    return

my_func()
print(x)
```

Concept Check!

What is printed by each of the following?

1.
```
x = 0
def my_func():
    x = 1
    print(x)
    return

my_func()
```

1

2.
```
x = 0
def my_func():
    x = 1
    return

my_func()
print(x)
```

Concept Check!

What is printed by each of the following?

1.
   ```
   x = 0
   def my_func():
       x = 1
       print(x)
       return

   my_func()
   ```

   ```
   1
   ```

2.
   ```
   x = 0
   def my_func():
       x = 1
       return

   my_func()
   print(x)
   ```

   ```
   0
   ```

Function I/O

Function I/O

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```

Function I/O

parameters

↓ ↓

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```

Function I/O

```
          parameters
            ↓  ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y  ←return statement
```

Function I/O

```
        parameters
            ↓  ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y  ←return statement
```

```
out1, out2 = my_func(3, 2)
```

Function I/O

```
        parameters
            ↓  ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y  ←return statement

return values      arguments
  ↓    ↓             ↓  ↓
out1, out2 = my_func(3, 2)
```

Function I/O

```
        parameters
            ↓  ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y  ←return statement

return values     arguments
  ↓    ↓             ↓  ↓
out1, out2 = my_func(3, 2)
print(out1) # prints 5
```

Function I/O

```
        parameters
            ↓  ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y  ←return statement

return values          arguments
  ↓    ↓                 ↓  ↓
out1, out2 = my_func(3, 2)
print(out1) # prints 5
print(out2) # prints 1
```

Algebra Functions

Algebra Functions

$$f(x) = 3x^2 + 2x + 1$$

Algebra Functions

$$f(x) = 3x^2 + 2x + 1$$

$$f(2)$$

Algebra Functions

$$f(x) = 3x^2 + 2x + 1$$

$$f(2) = 17$$

Mini-program

Mini-program

```
          parameters
            ↓   ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y   ←return statement
```

Mini-program

parameters

↓ ↓

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```

←*return statement*

parameters = inputs

Mini-program

```
            parameters
               ↓  ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y  ←return statement
```

parameters = inputs

body = calcualtions

Mini-program

```
        parameters
            ↓   ↓
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y   ←return statement
```

parameters = inputs

body = calcualtions

return values = prints

Familiar Functions

Familiar Functions

We've already been using several functions:

Familiar Functions

We've already been using several functions:

- `print(x, y, ...)` – receives 0 or more in inputs, returns nothing

Familiar Functions

We've already been using several functions:

- `print(x, y, ...)` – receives 0 or more in inputs, returns nothing
- `input(x)` – receives input message `x`, returns user's input

Familiar Functions

We've already been using several functions:

- `print(x, y, ...)` – receives 0 or more in inputs, returns nothing
- `input(x)` – receives input message `x`, returns user's input
- `str(x)/int(x)/float(x)` – receives variable `x`, returns converted value of `x`

Familiar Functions

We've already been using several functions:

- `print(x, y, ...)` – receives 0 or more in inputs, returns nothing
- `input(x)` – receives input message `x`, returns user's input
- `str(x)/int(x)/float(x)` – receives variable `x`, returns converted value of `x`
- `range(n)` – receives input `n`, returns a generator

Familiar Functions

We've already been using several functions:

- `print(x, y, ...)` – receives 0 or more in inputs, returns nothing
- `input(x)` – receives input message `x`, returns user's input
- `str(x)/int(x)/float(x)` – receives variable `x`, returns converted value of `x`
- `range(n)` – receives input `n`, returns a generator

Built-in/intrinsic functions

Pass-by-Reference

```
def ref_func(x):
    x = x + 1
    return

y = 1
ref_func(y)

y =  ?
```

Pass-by-Reference

```
def ref_func(x):
    x = x + 1
    return

y = 1
ref_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	00000001
00000011	10101110
⋮	⋮
11111111	00100000

Pass-by-Reference

```
def ref_func(x):
    x = x + 1
    return

y = 1
ref_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y , **x**	00000001
00000011	10101110
⋮	⋮
11111111	00100000

Pass-by-Reference

```
def ref_func(x):
    x = x + 1
    return

y = 1
ref_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y , x	**00000010**
00000011	10101110
⋮	⋮
11111111	00100000

Pass-by-Reference

```
def ref_func(x):
    x = x + 1
    return

y = 1
ref_func(y)

y =  ?
```

Memory:

Address	Contents
00000000	01101011
00000001	11001100
y	**00000010**
00000011	10101110
⋮	⋮
11111111	00100000

Pass-by-Reference

```
def ref_func(x):
    x = x + 1
    return

y = 1
ref_func(y)

y =  2
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	**00000010**
00000011	10101110
⋮	⋮
11111111	00100000

Pass-by-Value

```
def value_func(x):
    x = x + 1
    return

y = 1
value_func(y)

y =  ?
```

Pass-by-Value

```
def value_func(x):
    x = x + 1
    return

y = 1
value_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	00000001
00000011	10101110
⋮	⋮
11111111	00100000

Pass-by-Value

```
def value_func(x):
    x = x + 1
    return

y = 1
value_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	00000001
x	**00000001**
⋮	⋮
11111111	00100000

Pass-by-Value

```
def value_func(x):
    x = x + 1
    return

y = 1
value_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	00000001
x	**00000010**
⋮	⋮
11111111	00100000

Pass-by-Value

```
def value_func(x):
    x = x + 1
    return

y = 1
value_func(y)

y =  ?
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	00000001
00000011	00000010
⋮	⋮
11111111	00100000

Pass-by-Value

```
def value_func(x):
    x = x + 1
    return

y = 1
value_func(y)

y =  1
```

Memory:

Address	**Contents**
00000000	01101011
00000001	11001100
y	00000001
00000011	00000010
⋮	⋮
11111111	00100000

Tradeoffs

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

- Pass-by-reference is more efficient

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

- Pass-by-reference is more efficient
- Pass-by-value is safer and easier for programmers to use

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

- Pass-by-reference is more efficient
- Pass-by-value is safer and easier for programmers to use

Python uses a bit of both.

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

- Pass-by-reference is more efficient
- Pass-by-value is safer and easier for programmers to use

Python uses a bit of both.

Behaves more like pass-by-value in most cases.

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

- Pass-by-reference is more efficient
- Pass-by-value is safer and easier for programmers to use

Python uses a bit of both.

Behaves more like pass-by-value in most cases.

Don't try to reassign parameter values

Tradeoffs

Some programming languages use pass-by-reference and others use pass-by-value

- Pass-by-reference is more efficient
- Pass-by-value is safer and easier for programmers to use

Python uses a bit of both.

Behaves more like pass-by-value in most cases.

Don't try to reassign parameter values (until we get to OOP)

Concept Check!

What is printed by the following code?

1.
   ```
   def f(x):
       y = x * x
       return y

   print(f(5))
   ```

2.
   ```
   def f(x, y, z):
       a = x + y + z
       b = x * y * z
       return a, b

   x = 3
   out1, out2 = f(2, x, 4)
   print(out1)
   print(out2)
   ```

Concept Check!

What is printed by the following code?

1.
```
def f(x):
    y = x * x
    return y

print(f(5))
```

25

2.
```
def f(x, y, z):
    a = x + y + z
    b = x * y * z
    return a, b

x = 3
out1, out2 = f(2, x, 4)
print(out1)
print(out2)
```

Concept Check!

What is printed by the following code?

1.
```
def f(x):
    y = x * x
    return y

print(f(5))
```

25

2.
```
def f(x, y, z):
    a = x + y + z
    b = x * y * z
    return a, b

x = 3
out1, out2 = f(2, x, 4)
print(out1)
print(out2)
```

9

Concept Check!

What is printed by the following code?

1.
```
def f(x):
    y = x * x
    return y

print(f(5))
```

25

2.
```
def f(x, y, z):
    a = x + y + z
    b = x * y * z
    return a, b

x = 3
out1, out2 = f(2, x, 4)
print(out1)
print(out2)
```

9
24

Procedural Programming

Procedural Programming

```
statement1
statement2
statement3
    ⋮
```

Procedural Programming

```
statement1
statement2
statement3
    ⋮
statement1000
```

Procedural Programming

```
statement1
statement2
statement3
    ⋮
statement1000
```

- Hard to test and debug!

Procedural Programming

```
statement1
statement2
statement3
    ⋮
statement1000
```

- Hard to test and debug!
- What if someone wants to use just part of your code?

Procedural Programming

```
statement1
statement2
statement3
    ⋮
statement1000
```

- Hard to test and debug!
- What if someone wants to use just part of your code?
- What if you need to change something small on line 500?

Procedural Programming

Procedural Programming

```
def func1(x, y):
    statement1
    statement2
        ⋮
    return z

def func2():
    statement50
    statement51
        ⋮
    return

    ⋮
```

Procedural Programming

```
def func1(x, y):
    statement1
    statement2
        ⋮
    return z

def func2():
    statement50
    statement51
        ⋮
    return

    ⋮
```

```
a few input statements
out1 = func1(input1,
input2)
func2()
out2 = func3(out1)
    ⋮
```

Procedural Programming

```
def func1(x, y):
    statement1
    statement2
        ⋮
    return z
def func2():
    statement50
    statement51
        ⋮
    return

    ⋮
```

```
a few input statements
out1 = func1(input1,
input2)
func2()
out2 = func3(out1)
    ⋮
print(func10(out9, out10))
```

Modularity

Modularity

Procedural programs embody the engineering principle of *modularity*:

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems
- Build individual modules to solve each smaller problem

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems
- Build individual modules to solve each smaller problem
- Assemble modules to build full solution

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems
- Build individual modules to solve each smaller problem
- Assemble modules to build full solution
- Individual modules should have a standardized interface, so that they are easily interchangable or replacable