Fairfield UNIVERSITY

CPSC 1101
Introduction to Computing

School of Engineering & Computing
Dept. of Computer Science & Engineering
Dr. Sidike Paheding

Functions

- Function is a named sequence of statements that performs a computation/task.
- We can call a library of functions called a **module**
  - import math
- We can add parameters/arguments to functions to increase its dynamic functionality
  - add(a, b)
- Overloading a function will also increase its dynamic functionality
  - add(a, b)
  - add(a, b, c)

Functioning in Python

```
pi = [3,1,4,1,5,9]
```

- What is **len(pi)** ?

6

```
Q = [ 'pi', "isn't", [4,2] ]
```

- What is **len(Q)** ?

3

- Len is a built-in *function*

Functioning in Python (cont'd)

```
# my own function!
def dbl( x ):
    """
    Accepts an argument, doubles it,
    and returns the double value.
    """
    print(" double the value of x ")
    return 2*x
```

comment for other ***coders***

documentation string for all ***users***

Python's keywords

Functioning across disciplines

structure

$$g(x) = x^{100}$$

procedure

```
def g(x):
    return x**100
```

Math's function statement

defined by ***what it is***

+ what follows ***logically***

CS's function statement

defined by ***what it does***

+ what follows ***behaviorally***

Giving names to data *helps f'ns*

```
def flipside(s):
    print(" flipside(s): swaps s's sides!
            input s: a string ")
    x = int(len(s)/2)
    return s[x:] + s[:x]
```

Key idea: store values locally

return vs. print

```
def dbl(x):
    """ dbls x? """
    return 2*x

>>> ans = dbl(20)
print(ans) >> 40
```

```
def dblPR(x):
    """ dbls x? """
    print(2*x)

>>> ans = dblPR(20)
print(ans) >> None
```

What's the difference ?!

return vs. print

```
def dbl(x):
    """ dbls x? """
    return 2*x

>>> ans = dbl(20)+2
```

yes!

```
def dblPR(x):
    """ dbls x? """
    print(2*x)

>>> ans = dblPR(20)+2
```

ouch!

print display output on the screen...

return yields the function call's ***value*** ...

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

*they **stack**.*

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

stack frame

call: `demo(15)`

local variables:

```
x = 15
y = 5
z = ?????
```

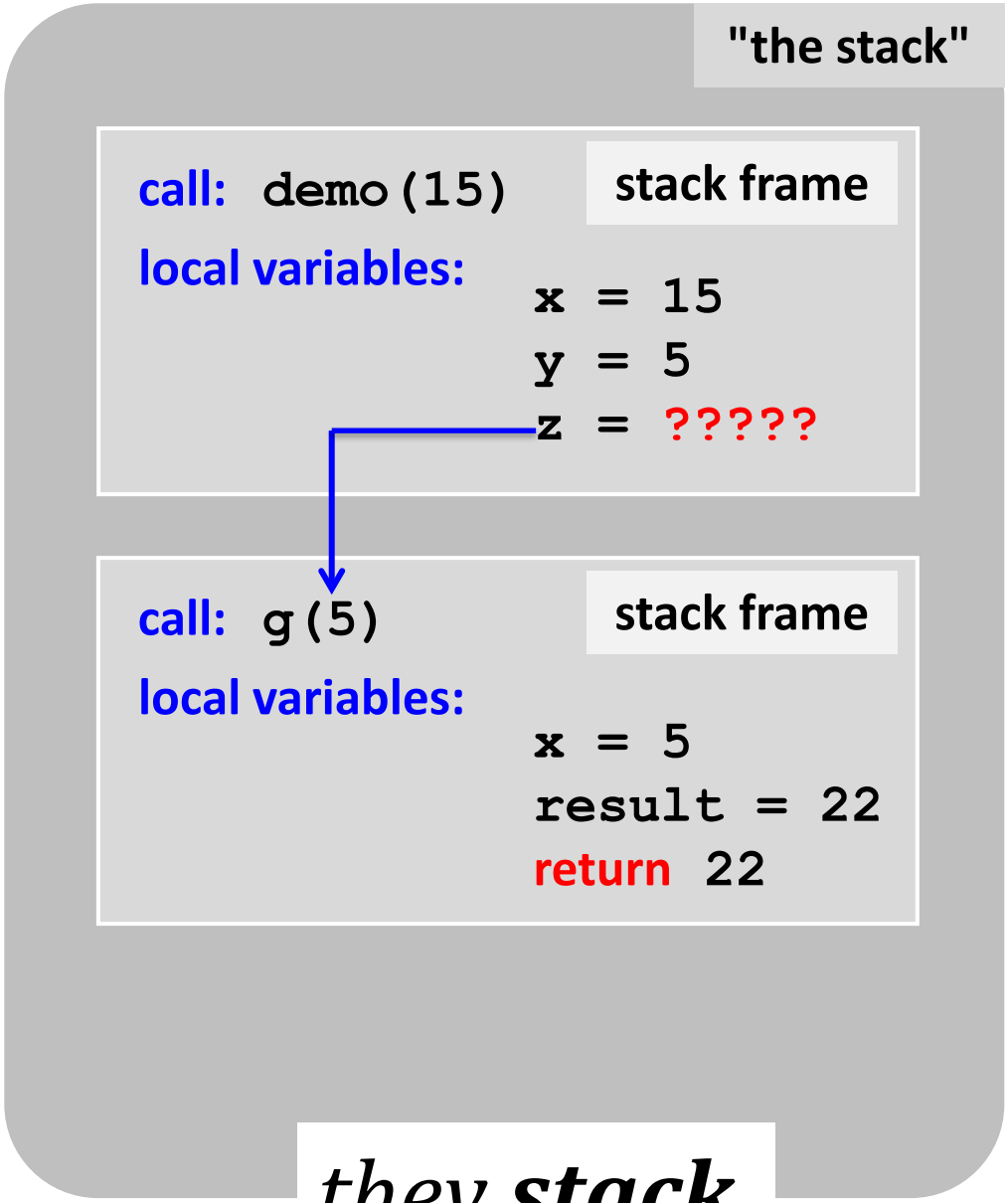
*they **stack**.*

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```



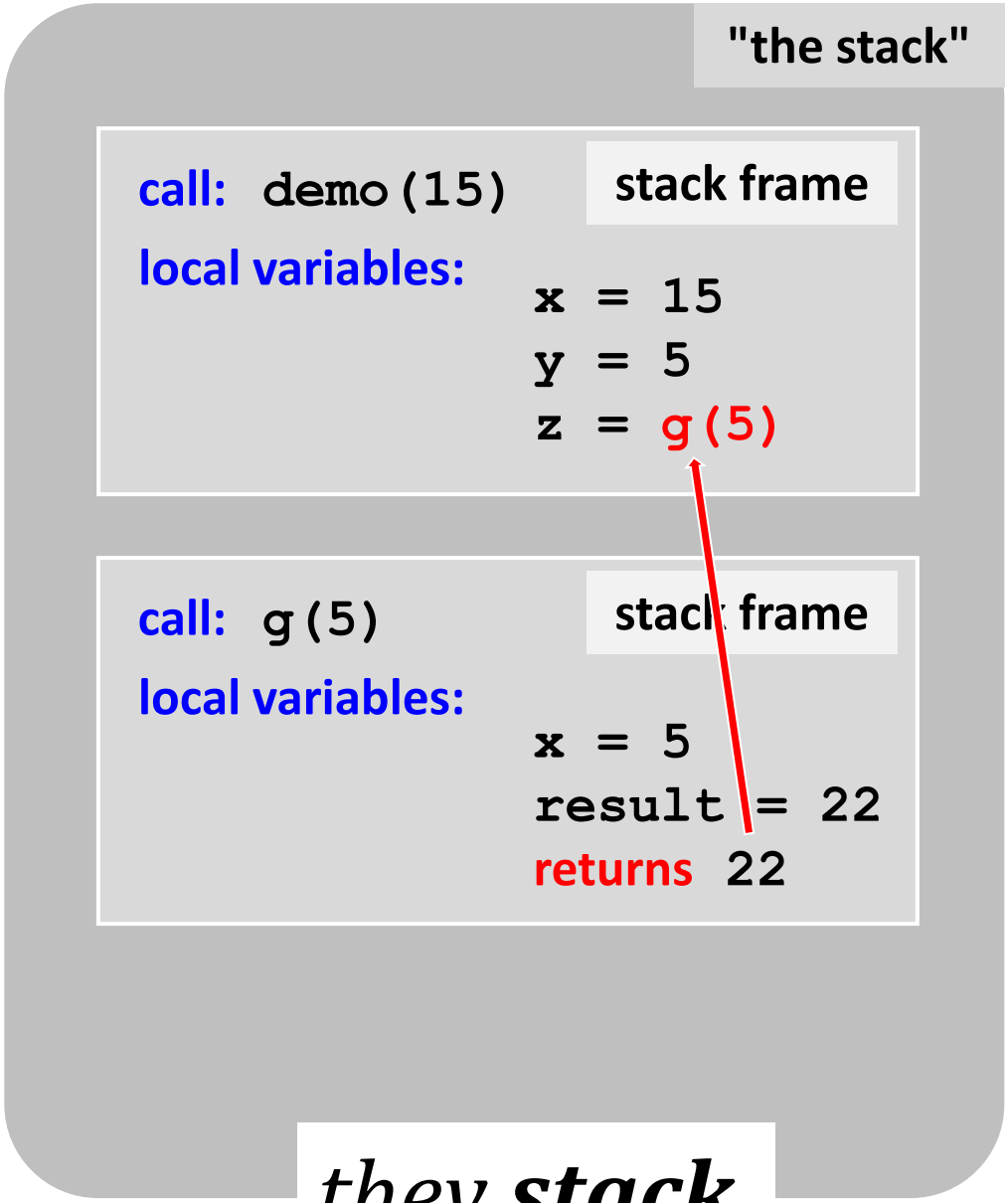
*they **stack**.*

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```



*they **stack**.*

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

stack frame

call: `demo(15)`

local variables:

```
x = 15
y = 5
z = 22
```

*they **stack**.*

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

stack frame

call: `demo(15)`

local variables:

```
x = 15
y = 5
z = 22
```

return 42

they ***stack.***

How functions work...

15

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

42

output

"the stack"

afterwards, the stack is empty..., but ready if another function is called

*they **stack**.*

How functions work...

2

what's **f(2)** ?

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

How functions work...

1

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stack frame

call: `f(1)`

local variables:

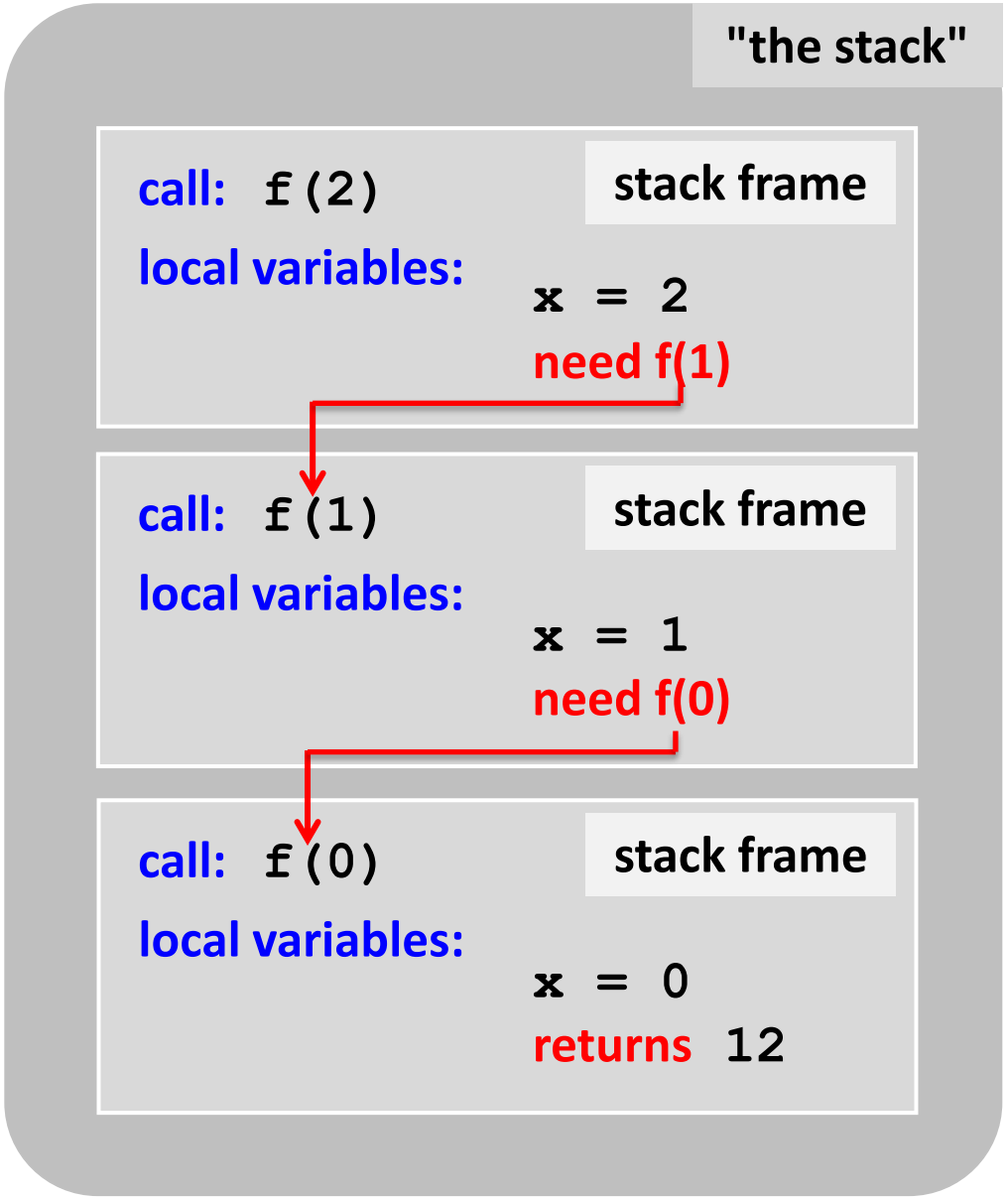
`x = 1`

need f(0)

How functions work...

0

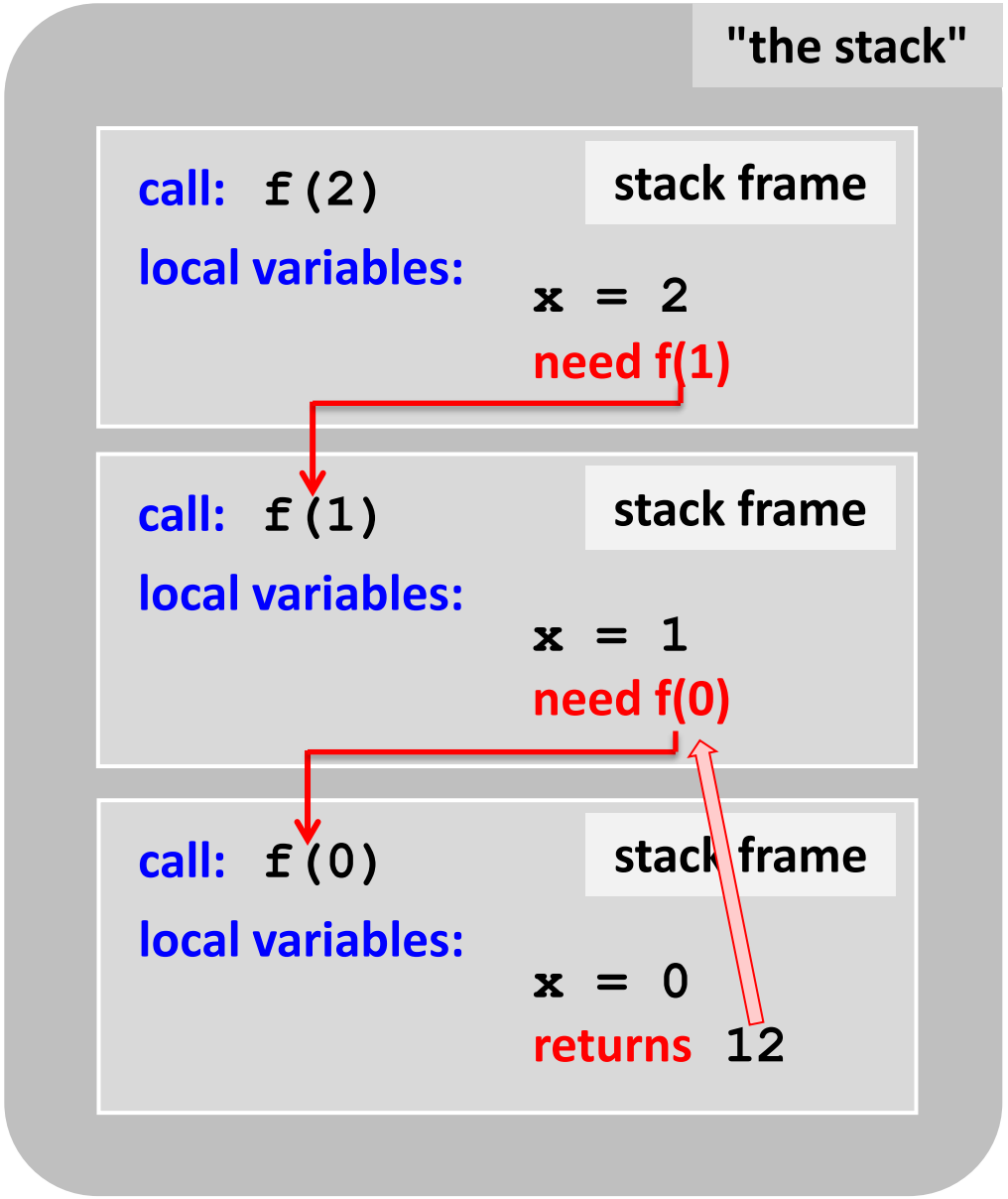
```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```



How functions work...

0

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```



How functions work...

1

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stack frame

call: `f(1)`

local variables:

`x = 1`

f(0) = `12`

result =

How do we compute the result?

How functions work...

1

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stack frame

call: `f(1)`

local variables:

`x = 1`

f(0) = `12`

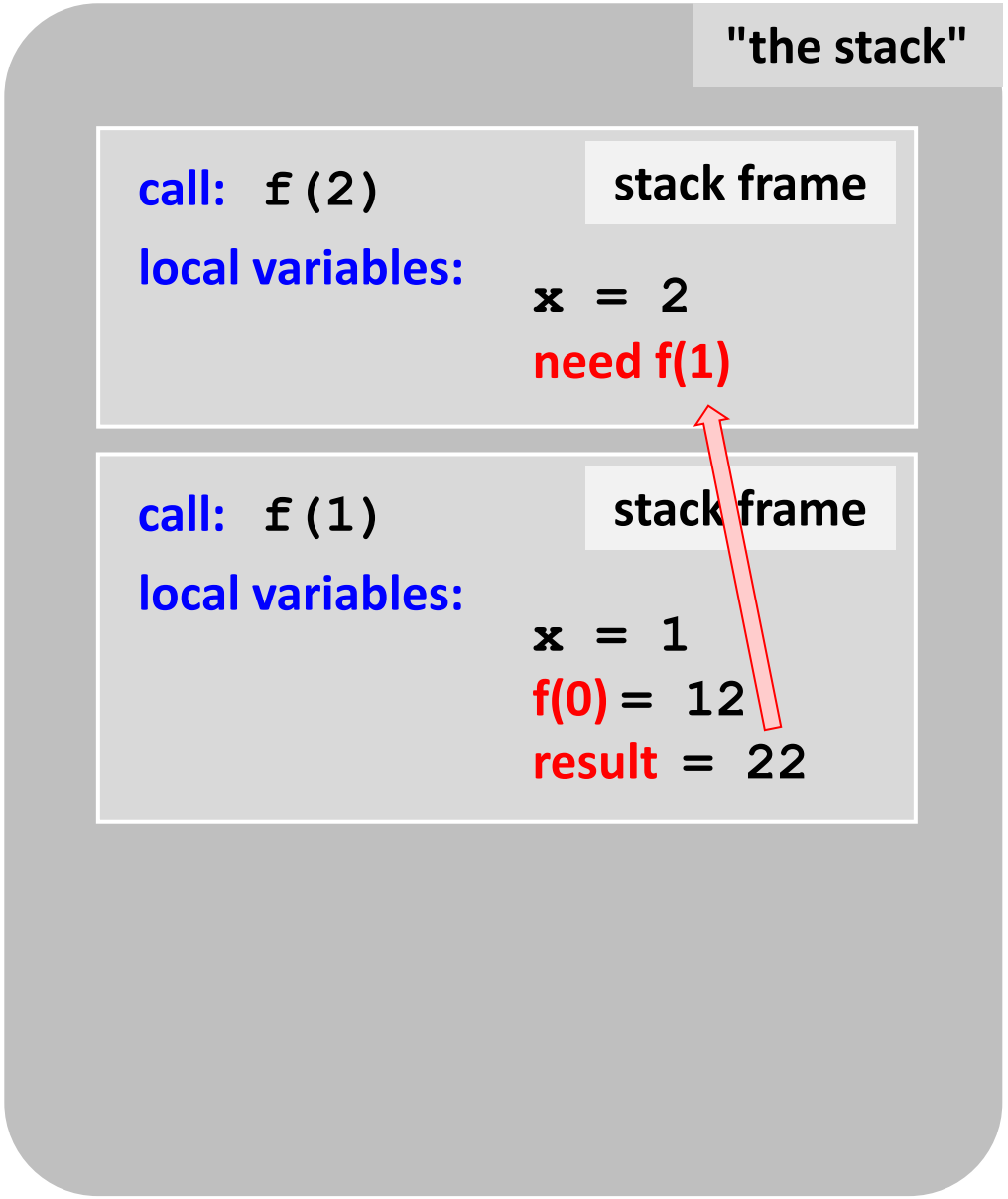
result = `22`

Where does that result go?

How functions work...

1

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```



How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) = 22

result =

What's *this* return value?

How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) = `22`

result = `42`

which then gets returned...

How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) `= 22`

result `= 42`

the result then gets returned...

How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

42

output

"the stack"

again, the stack is empty, but ready if another function is called...

*functions **stack**.*

How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

42

output

"the stack"

again, the stack is empty, but ready if another function is called...

Functions are like cells: software's building blocks...

*... each one, a **self-contained** computational unit!*

*functions **stack**.*

ICE 1

1. What is the output of the Python code on the right?

2. Given the numbers 23 and 33 write 3 different functions to add them.
3. Create a function to swap values of two input variables.
4. Create a function that prints each item in a list one by one.

```python
def example(a):
    b = a * 2
    c = h(b)
    return c + b + a

def h(x):
    result = 3 * x - 1
    return result

print(example(5))
```

How to define and call a main() function

- When you use one or more functions in a program, it is a good practice to put all of the code for the program in functions.
- You put all of the code that isn’t in specific functions but in a main() function.
- The code in the main() function starts the operation of the program.

Two ways to call a main() function

1. Code a simple call statement (not recommended)

```
main()
```

2. Code a call statement within an if statement that checks if current module is main module

```
if __name__ == "__main__":  # if main module
    main()                  # call main() function
```

- The if __name__ == "__main__": is used to determine whether the current script is being run as the main program or if it is being imported as a module into another script.

What does the if __name__ == “__main__”: do?

- In Python, __name__ is a special built-in variable that represents the name of the module.
- If a Python script is executed directly (e.g., python script.py), __name__ is set to "__main__".
- If the script is imported as a module in another script (e.g., import script), __name__ is set to the module's name ("script").
- It checks whether the script is being run directly or being imported.
  - If it evaluates to True (i.e., the script is run directly), the code under this block will execute.
  - If the script is imported, this block will not run.

Example

- Run this code directly by calling `my_main_function.py`.
- Use `import my_main_function` in another script.

```python
# my_main_function.py

def greet():
    print("Hello from Python!")

if __name__ == "__main__":
    print("My function is being run directly")
    greet()
else:
    print("My function has been imported")
```

Lambda function

lambda arguments: expression

- **lambda**: Keyword to define a lambda function.
- **arguments**: Input parameters (can be multiple, separated by commas).
- **expression**: Operation or calculation that the lambda function performs. The result of this expression is automatically returned.

Example of a Lambda Function

```
add_10 = lambda x: x + 10
print(add_10(5))  # Output: 15

x = 5
add_10 = lambda y: y + x
print(add_10(5))  # Output: 10
```

Lambda with Multiple Arguments

```
multiply = lambda x, y: x * y
print(multiply(3, 4))  # Output: 12

operations = {
    "add": lambda x, y: x + y,
    "subtract": lambda x, y: x - y
}
print(operations["add"](10, 5))       #
Output: 15
print(operations["subtract"](10, 5))  #
Output: 5
```

Lambda with other functions

```
numbers = [1, 2, 3, 4, 5, 6]

# Use a lambda function with filter() to keep only
even numbers
even_numbers = list(filter(lambda x: x % 2 == 0,
numbers))
            >> [2, 4, 6]

# Use a lambda function with map() to square each
even number
squared_even_numbers = list(map(lambda x: x ** 2,
even_numbers))
            >> [4, 16, 36]
```

The map() function is used to apply a given function to every item of an iterable.

ICE 2

- Write a Python function that swaps a string's first and last letters.

1) Ask a user to enter a string
2) If a string has less than two letters, return the string itself.
3) If a string has at least two letters, swap a string's first and last letters, then return the string.

ICE 3

A Convert Temperatures program:

- The user enters a 1 or a 2 to indicate what type of conversion should be done followed by the number of degrees to be converted.
- The program displays the result.

A sample screen snapshot

```
MENU
1. Fahrenheit to Celsius
2. Celsius to Fahrenhit

Enter a menu option: 1
Enter degrees Fahrenheit: 99
Degrees Celsius:  37.22

Convert another temperature? (y/n): y

Enter a menu option: 2
Enter degrees Celsius: 23
Degrees Celsius:  73.4

Convert another temperature? (y/n): n

Bye!
```

ICE 4

Filtering and Modifying Product Prices Based on Category

- Suppose you have a list of products with categories and prices, and you want to:

1. Filter out only the products in the "Electronics" category.
2. Apply a 10% discount to these filtered products.

- Use the lambda function to solve this problem.

Fairfield UNIVERSITY

CPSC 1101
Introduction to Computing

School of Engineering & Computing
Dept. of Computer Science & Engineering
Dr. Sidike Paheding