Python is... `in`

I guess Python's the **in** thing



```
>>> 'i' in 'team'
False

>>> 'cs' in 'physics'
True

>>> 'i' in 'alien'
True

>>> 42 in [41,42,43]
True
```

```
>>> 3*'i' in 'alien'
False

>>> 42 in [[42], '42']
False
```

What is a function?

- Syntax point of view
  - A named sequence of statements

```
def name( parameters ):
    statement 1
    statement 2
    …
```

- Semantic point of view
  - Used to a big problem down to smaller chunks
  - Put together to solve a larger problem
    - Divide and conquer!

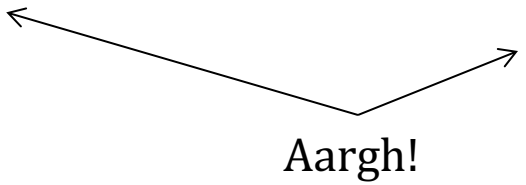
Giving names to data *helps functions*

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

Use variables!

```
def flipside(s):
    mid = len(s) // 2
    return s[mid:] + s[:mid]
```

```
def flipside(s):
    return s[len(s) // 2:] + s[:len(s) // 2]
```



Why would computers "prefer" the top version, too?

return vs. print

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

```
In[1]: ans = dbl(20)
```

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

```
In[1]: ans = dblPR(20)
```

What's the difference ?!

return > 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 changes pixels on the screen...

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

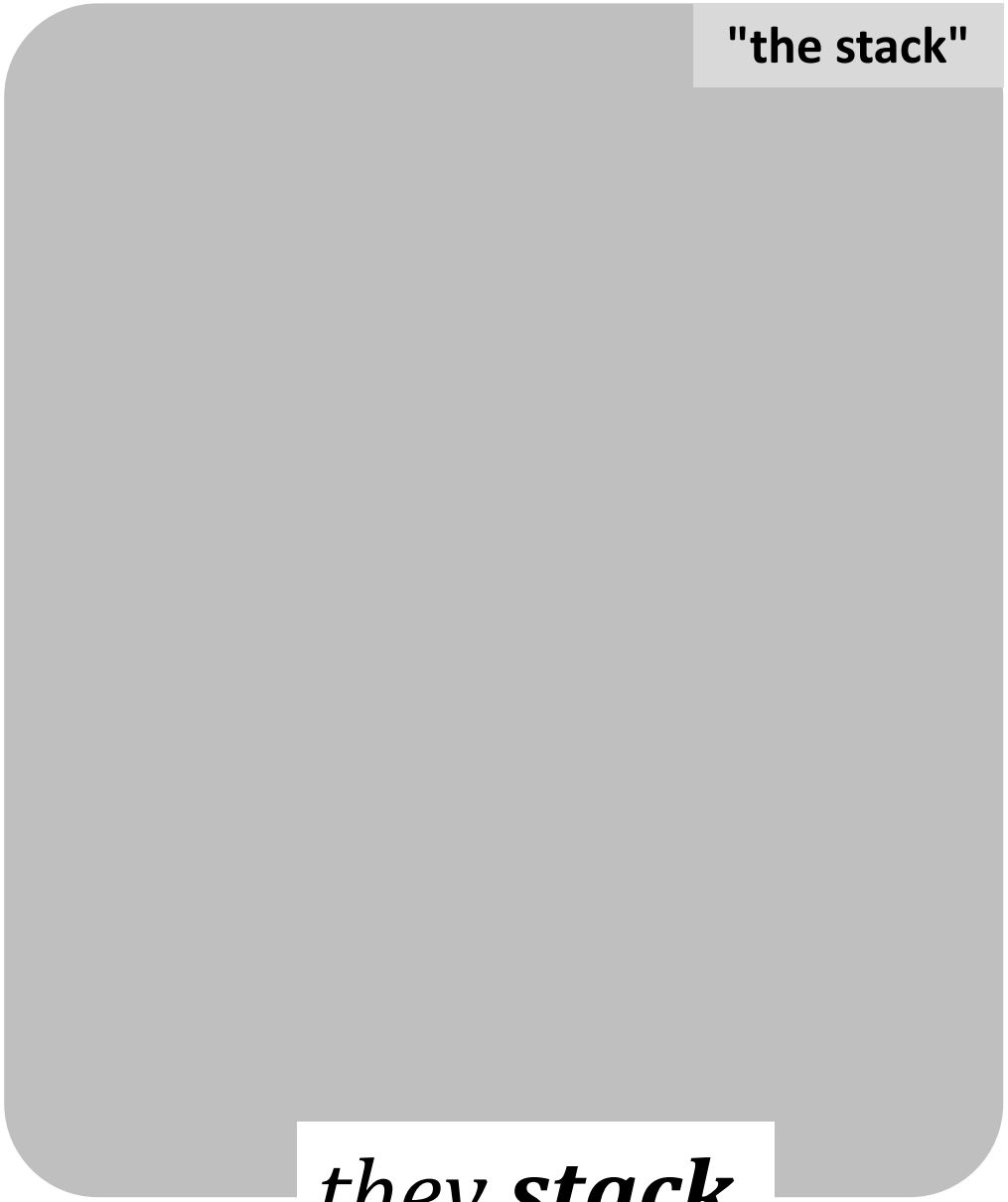
... which the shell then prints!

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 = ?????
```

*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"

call: `demo(15)` — stack frame

local variables:

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

call: `g(5)` — stack frame

local variables:

```
x = 5
result = 22
```

returns 22

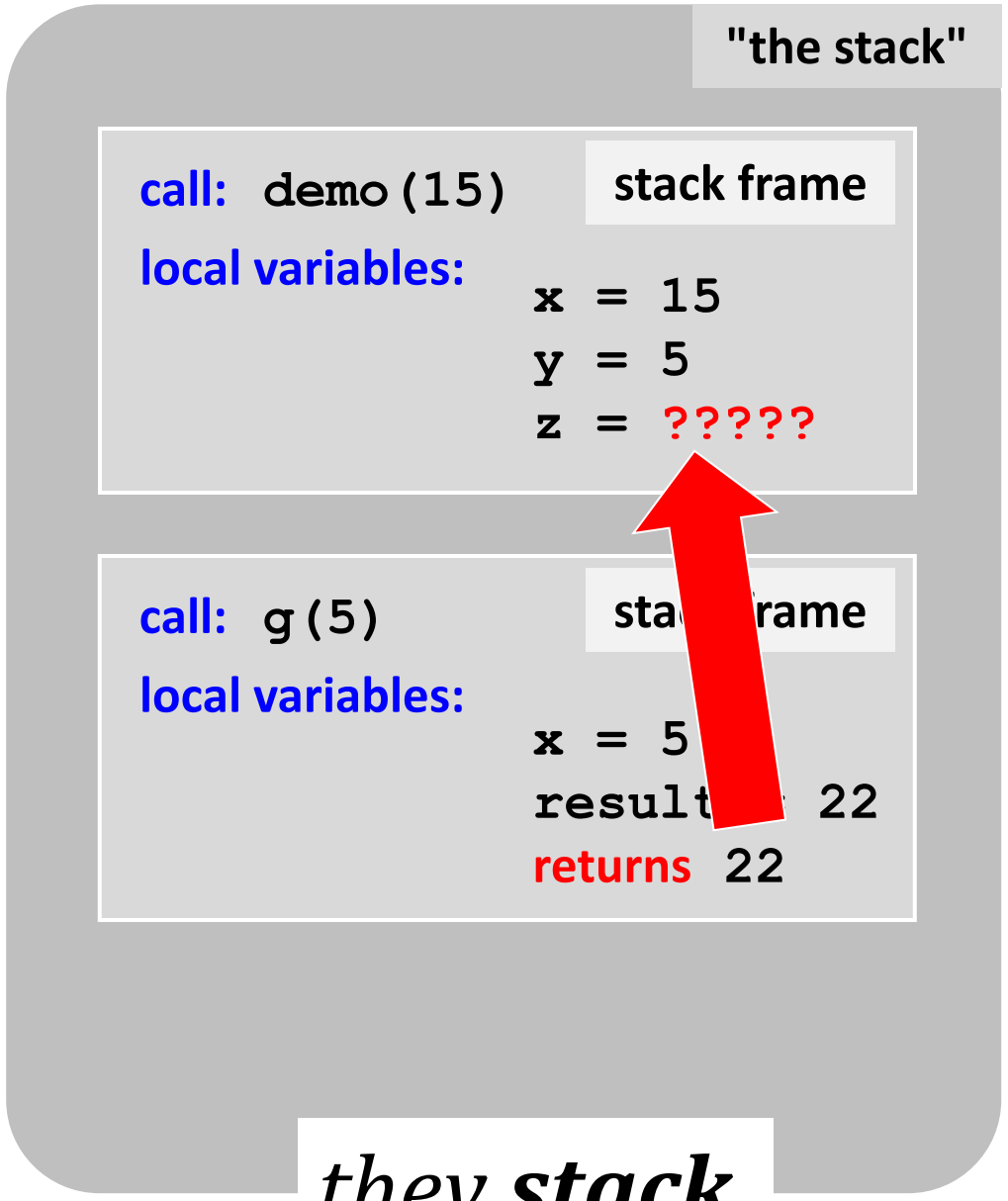
*they **stack**.*

How functions work...

15

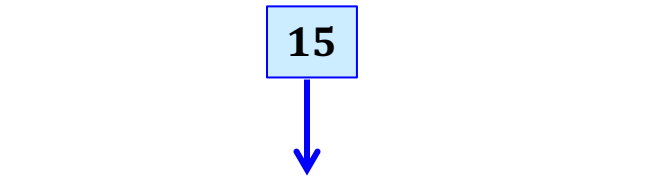
```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

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



*they **stack**.*

How functions work...




```
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
```

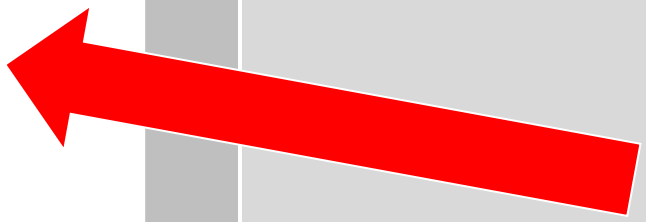
*they **stack**.*

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...

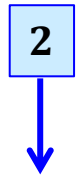
2

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

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

"the stack"

How functions work...




```
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"

call: `f(2)` — stack frame

local variables:

`x = 2`

need f(1)

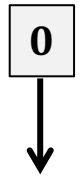
call: `f(1)` — stack frame

local variables:

`x = 1`

need f(0)

How functions work...




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

"the stack"

call: `f(2)` — **stack frame**

local variables:

`x = 2`

need f(1)

call: `f(1)` — **stack frame**

local variables:

`x = 1`

need f(0)

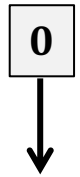
call: `f(0)` — **stack frame**

local variables:

`x = 0`

returns 12

How functions work...




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

"the stack"

call: `f(2)` — stack frame

local variables:

`x = 2`

need f(1)

call: `f(1)` — stack frame

local variables:

`x = 1`

need f(0)

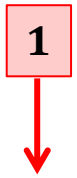
call: `f(0)` — stack frame

local variables:

`x = 0`

returns 12

How functions work...




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

"the stack"

call: `f(2)`

local variables:

stack frame

`x = 2`

need f(1)

call: `f(1)`

local variables:

stack frame

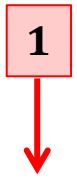
`x = 1`

f(0) = 12

result =

How do we compute the result?

How functions work...




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

"the stack"

call: `f(2)` — **stack frame**

local variables:

`x = 2`

need f(1)

call: `f(1)` — **stack frame**

local variables:

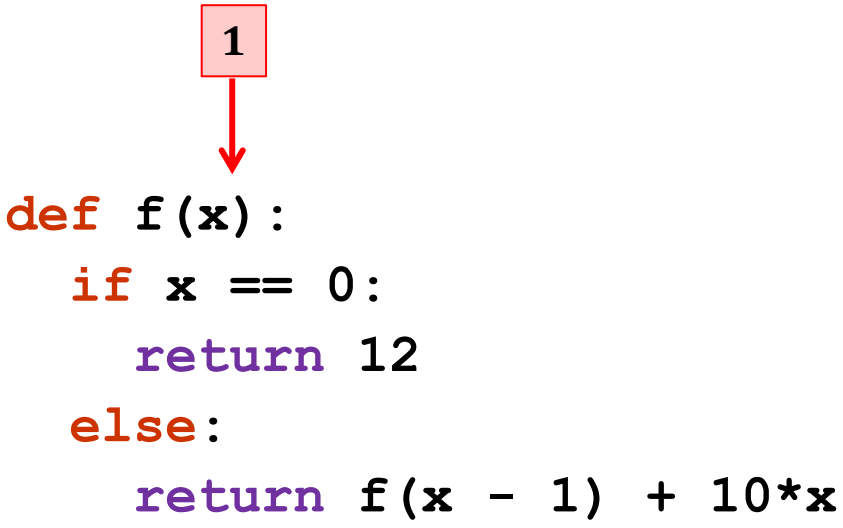
`x = 1`

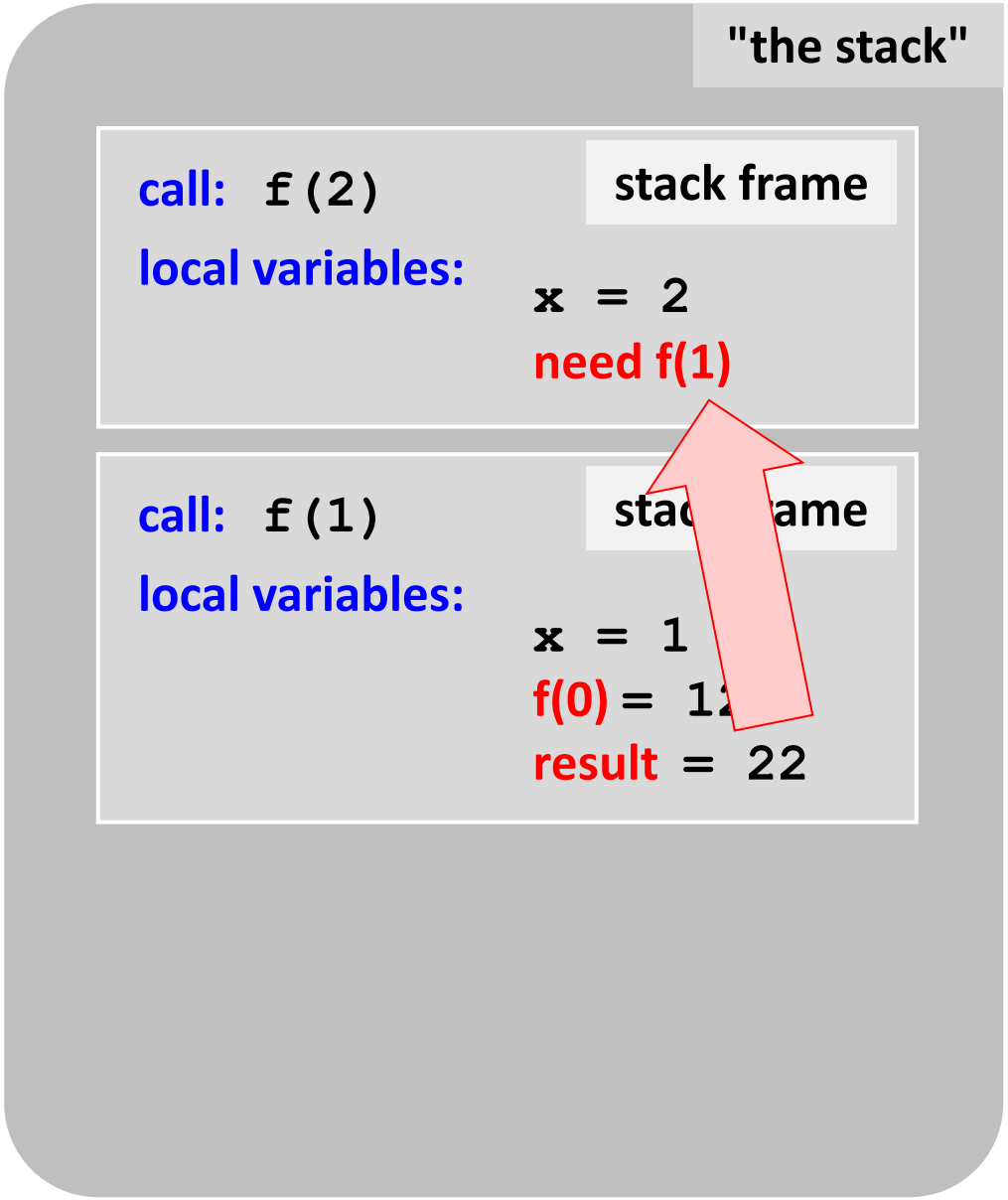
f(0) = 12

result = 22

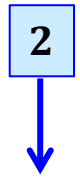
Where does that result go?

How functions work...





How functions work...




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

"the stack"

stack frame

call: `f(2)`

local variables:

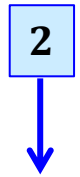
`x = 2`

f(1) = 22

result =

What's *this* return value?

How functions work...




```
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

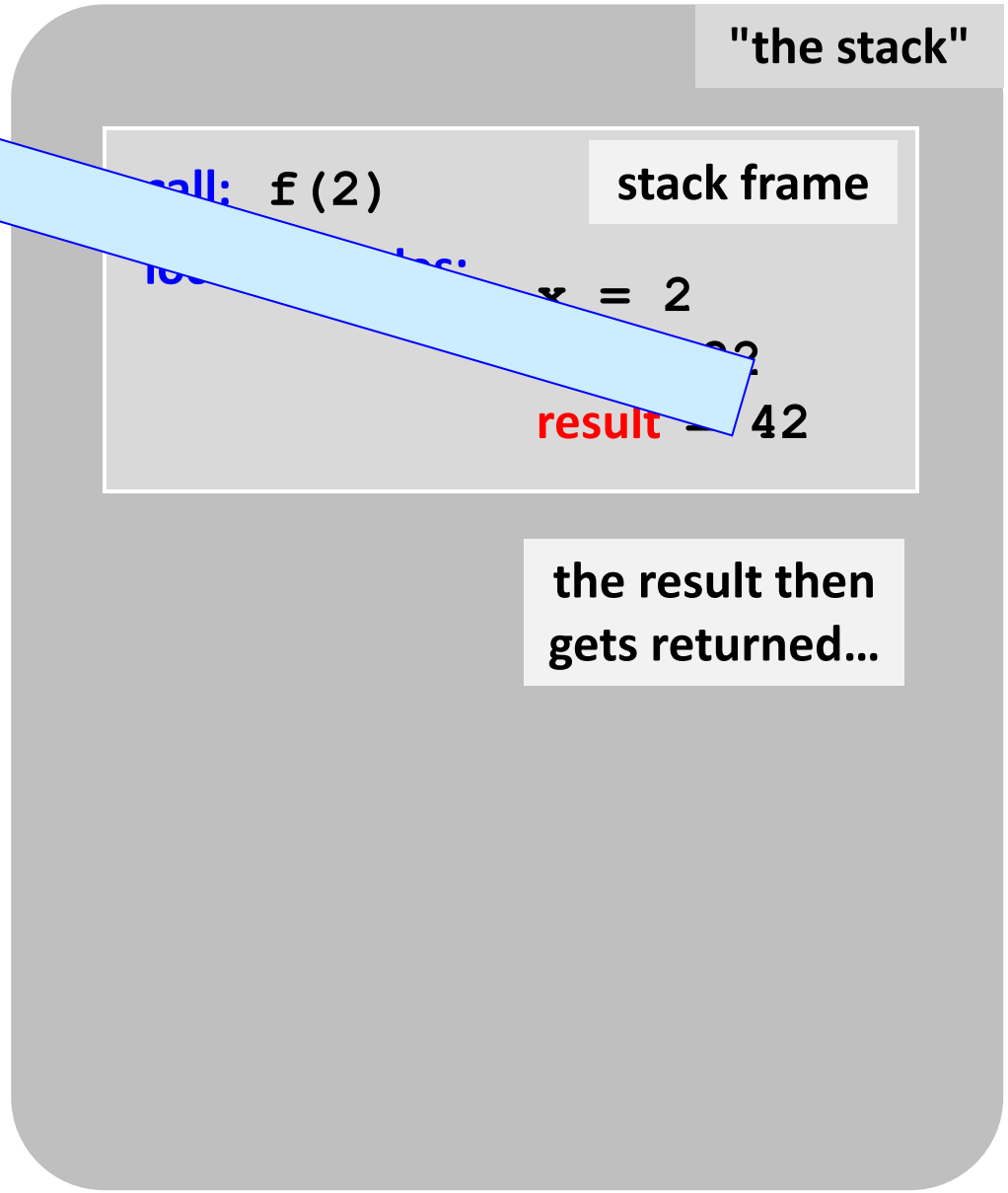
result = 42

which then gets returned...

How functions work...

2

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



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**.*