



CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 3 Name, Scope, Binding and Environment

LECTURE 3: PYTHON REFERENCE ENVIRONMENT

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Definition of names, binding, scope, and environment.
- Binding: Allocation of Memory
- Memory Models
- Scope Rules
- Reference Environments

Variable Names, Binding and Scope

SECTION 1

Name, Scope, and Binding

id, data lifetime, and data memory association

- A **name** is exactly what you think it is
 - Most names are identifiers
 - symbols (like '+') can also be names
- A **binding** is an association between two things, such as a name and the thing it names
- The **scope of a binding** is the part of the program (textually) in which the binding is active

Name, Scope, and Binding

- Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
 - language design time
 - program structure, possible type
 - language implementation time
 - I/O, arithmetic overflow, type equality (if unspecified in manual)

Static vs. Dynamic Binding

- **Binding**

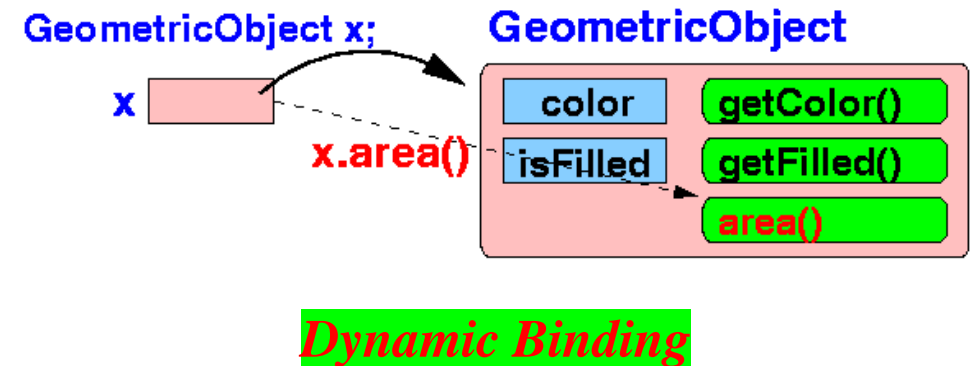
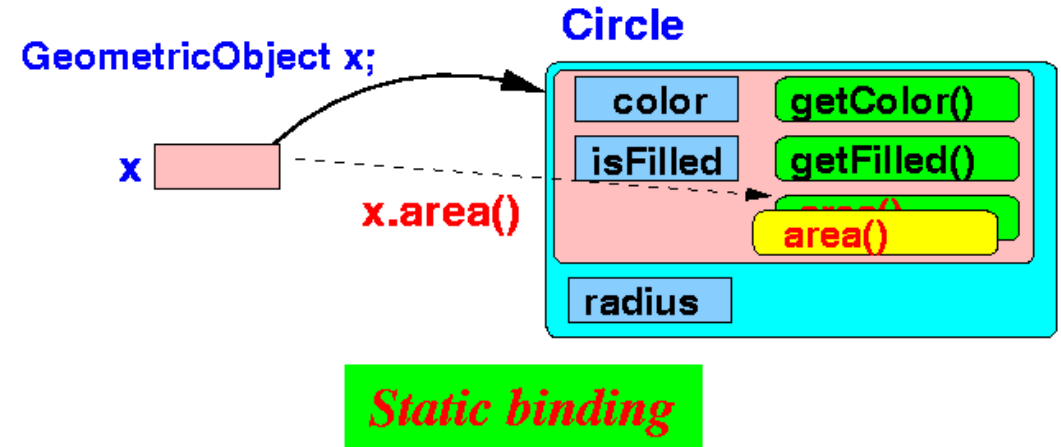
The determination of which method in the class hierarchy is to be used for a particular object.

- **Static (Early) Binding**

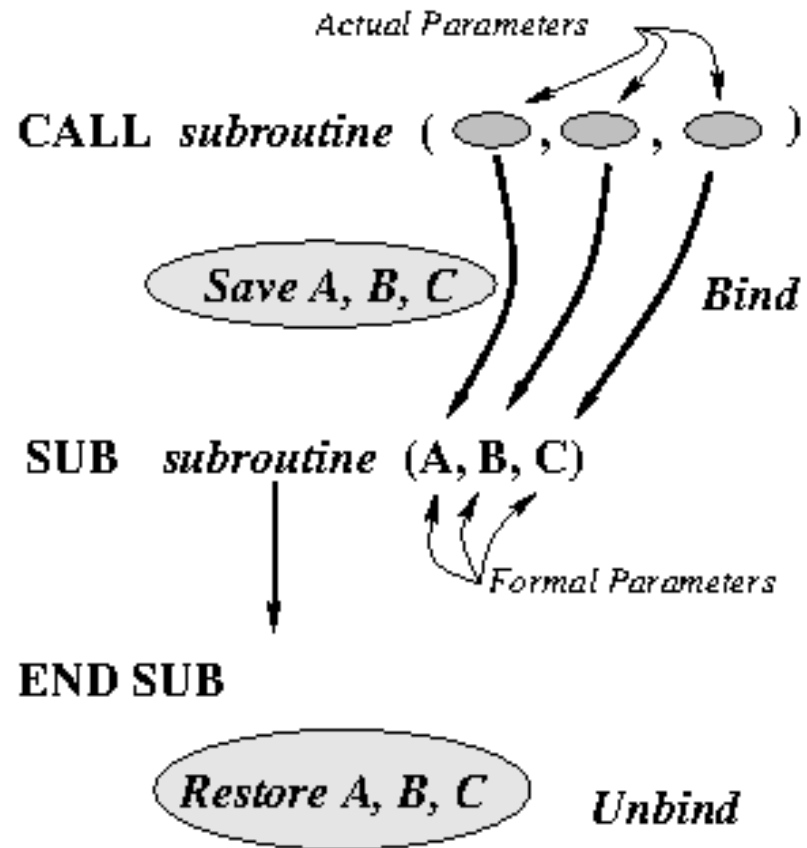
When the compiler can determine which method in the class hierarchy to use for a particular object.

- **Dynamic (Late) Binding**

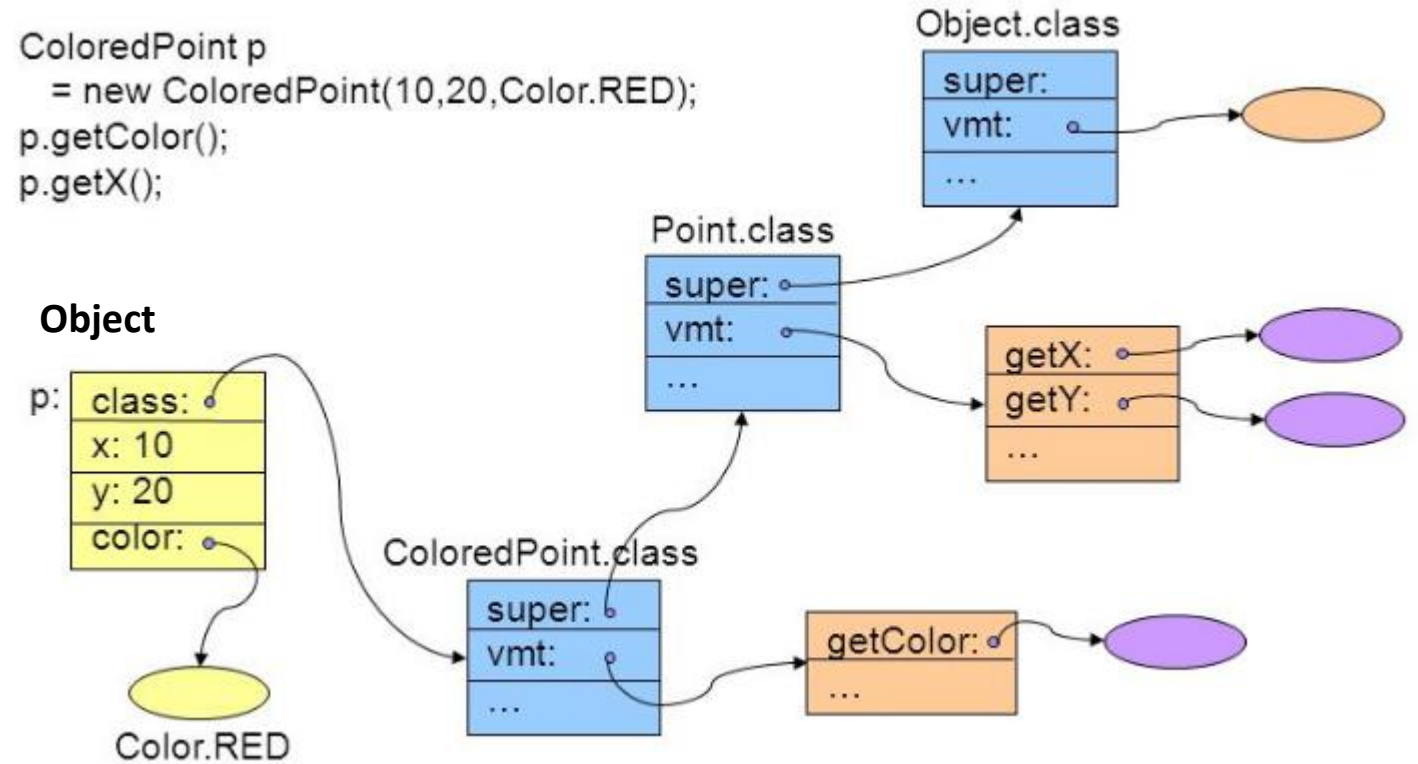
When the determination of which method in the class hierarchy to use for a particular object occurs during program execution.

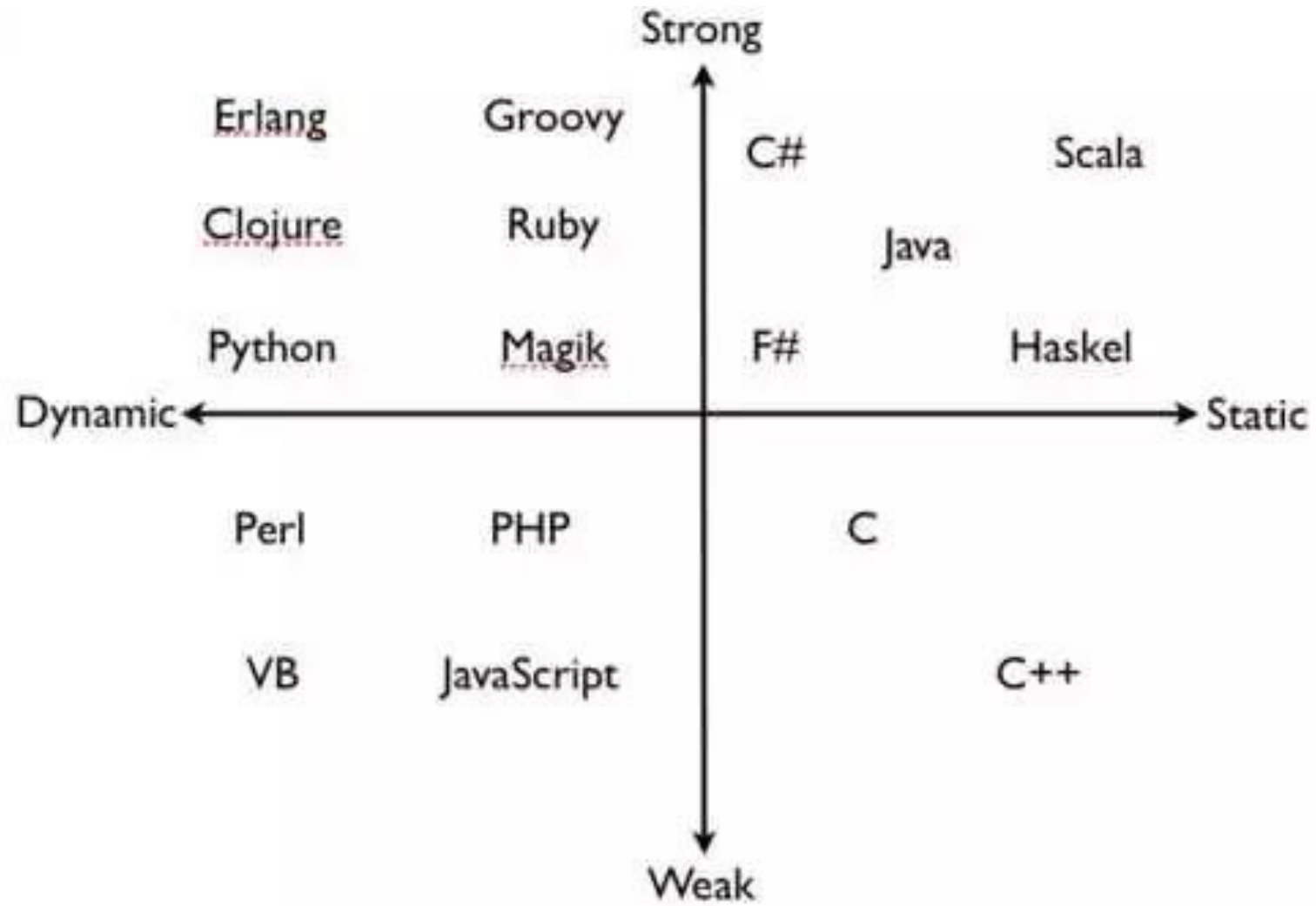


Parametric Binding



Polymorphic Method Binding





Python is Statically Scoped but Dynamic Binding

Dynamic Typed/Strong Typed:

- Python is a dynamically typed language. ... Python don't have any problem even if we don't declare the type of variable. It states the kind of variable in the runtime of the program. So, Python is a dynamically typed language.
- Strong typed.

Python is Statically Scoped but Dynamic Binding

Statically Scoped:

- The scope is static in Python as well as in C++. The difference between those languages is related to rules that define start and end of the scope of names.
- In C++ the variable `i` is considered as global before the local definition `int i=15;`

Python is Statically Scoped but Dynamic Binding

Dynamic Binding:

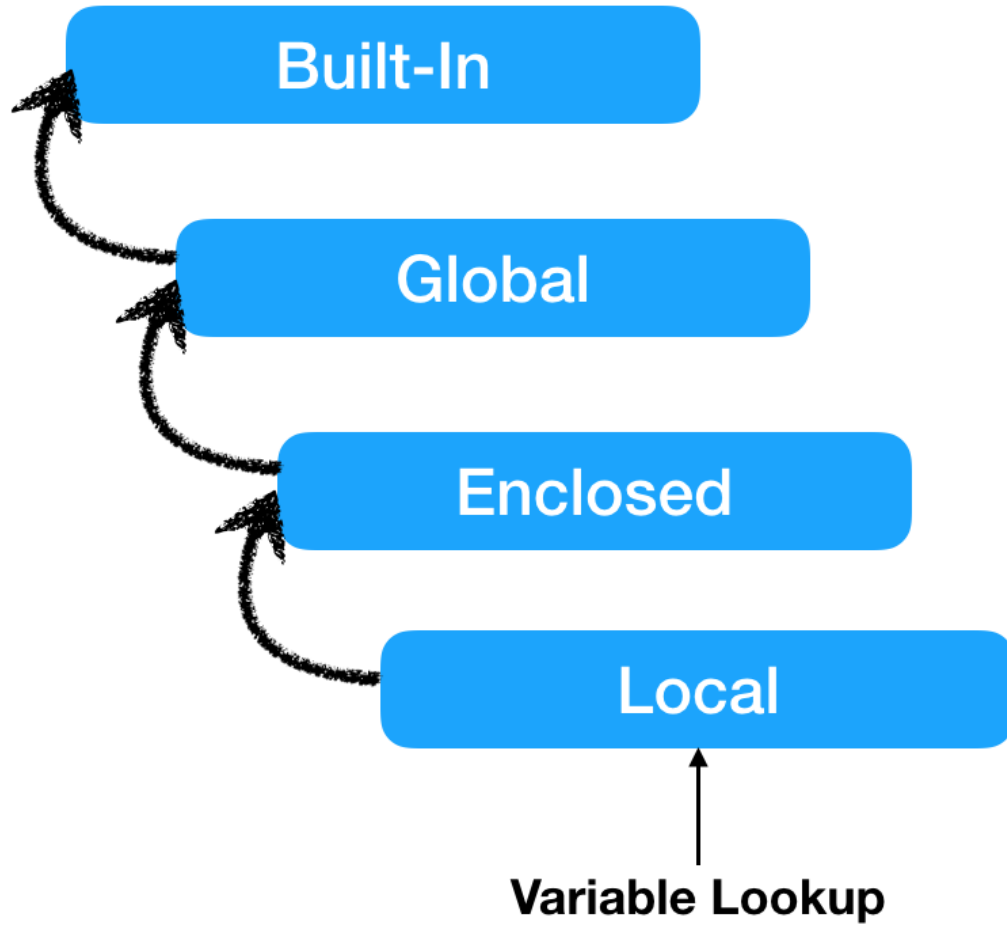
- Binding in Python is the process of giving an object a name - for instance when you do

```
the_string = 'Hello World'
```

- You are binding the name 'the_string' to a string object with the value 'Hello World'.
- In Python all of this happens at run-time. Dynamic binding is when this happens in some way based on user input or other data. An example is using setattr() to create a attribute or variable - you would normally use setattr because the name to be bound is itself stored as a variable.

Python Namespace

SECTION 2



LEGB Rule

Namespaces – Local and Global - Topics

- The purpose of Functions
- Global versus Local Namespaces
- The Program Stack
- How Python Evaluate Names

The Purpose of Functions

Functions make it easier for us to code solution to problems:

- **Modularity:** They allow us to break down complex problems that require complex programs into small, simple, self-contained pieces. Each small piece can be implemented, tested and debugged independently.
- **Abstraction:** A function has a name that should clearly reflects what it does. That action can then be performed by calling the function by name, abstracting what the function does from how it does it.
- **Code reuse:** Code that may be used multiple times should be packaged in a function. The code only needs to be written once. And any time that it needs to be modified, extended or debugged, the changes need to be made only one.

Local Variables hide What goes on inside a function

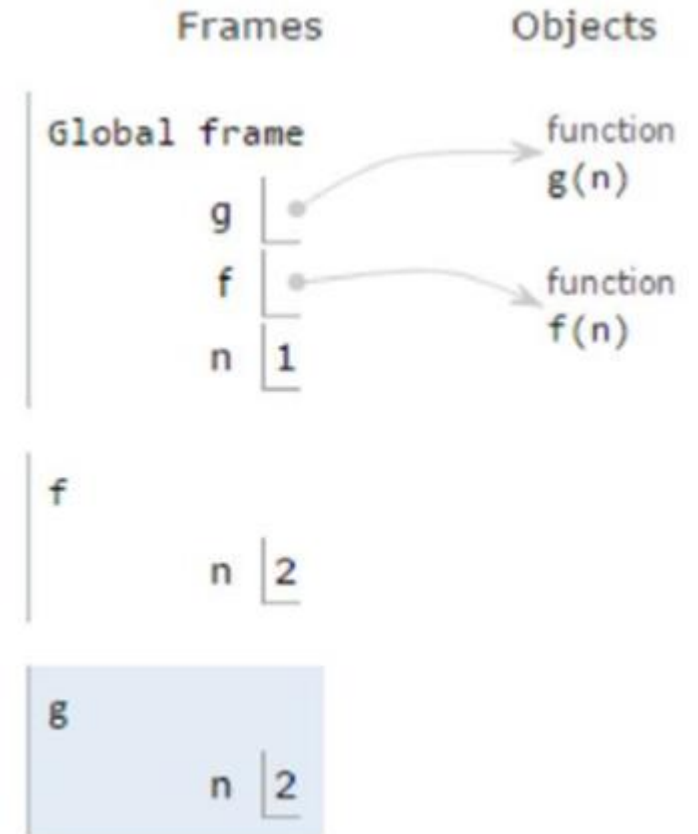
- Enter this code in [pythontutor](#) and trace its execution.
- First, x and y are created in the global frame.
- While double executes, local x and y are created with their own values. These local variables cease to exist when the function exits.
- Separating what happens inside a function from what happens outside is called encapsulation.

```
def double(y):  
    x = 2  
    y *= x  
    print('inside double', 'x = ', x, 'y = ', y)  
  
x, y = 3, 4  
print('outside double', 'x = ', x, 'y = ', y)  
double(y)  
print('after double', 'x = ', x, 'y = ', y)
```


Function Namespace

- Every execution of a function creates a local namespace that contains any local variables.
- Note that there are several active values of `n`, one in each namespace. How are all the namespaces managed in Python? Which line does Python return to?

```
def g(n):  
    print('Start g')  
    n += 1  
    print('n = ', n)  
  
def f(n):  
    print('Start f')  
    n += 1  
    print('n = ', n)  
    g(n)  
  
n = 1  
print('Outside a function, n = ', n)  
f(n)
```



Changed to 3 when called

Scope and Global versus Local Namespace

Every function call has its own (local) namespace

- This namespace is where names defined during the execution of the function (e.g, local variables) live.
- This namespace comes into existence when the function is called. It goes out of existence when the function exits (returns).

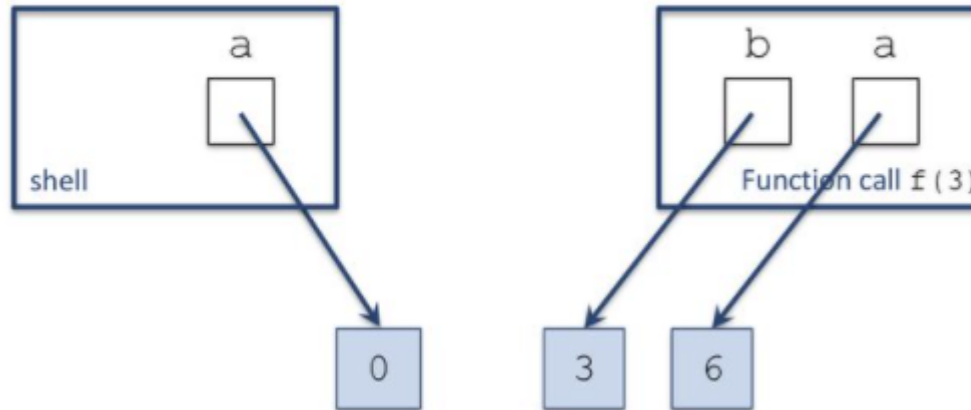
Every name in a python program has a scope

- This applies to the name is of a variable, a function, a class, ...
- Outside its scope, the name does not exist. Any reference to it will result in an error.
- **Names created in the interpreter shell or in a module and outside of any function have global scope.**

Example: variable with local scope

```
def f(b): # f has global scope, b has local scope
    a = 6 # this a has scope local to this call of function f()
    return a*b # this a is the local a

a = 0 # this a has global scope
print('f(3) = ', f(3))
print('a = ', a) # global a is still 0
```

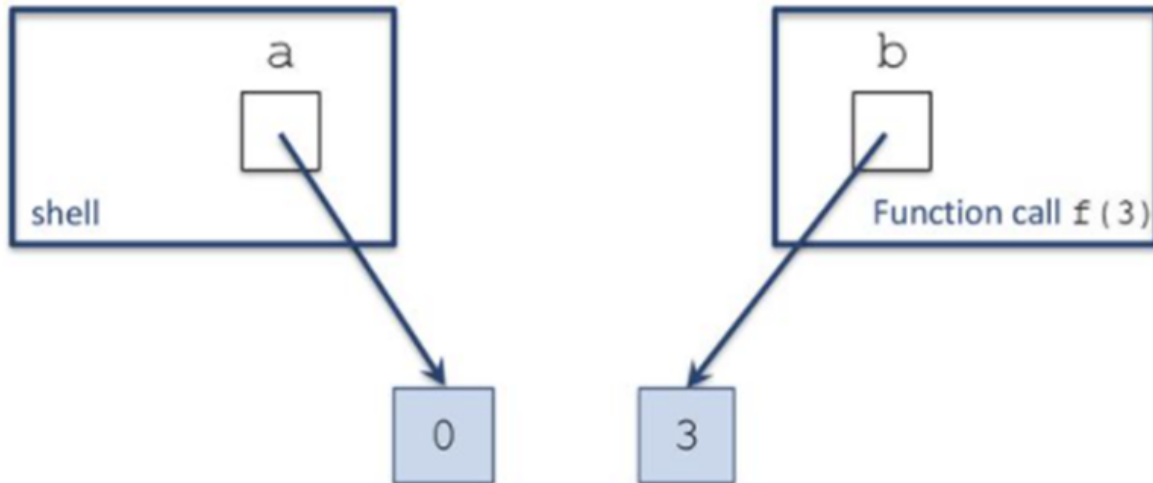


```
>>> === RESTART ===
>>>
f(3) = 18
a = 0
>>>
```

Example: variable with global scope

```
def f(b):          # f has global scope, b has local scope
    return a*b     # this a is the global a

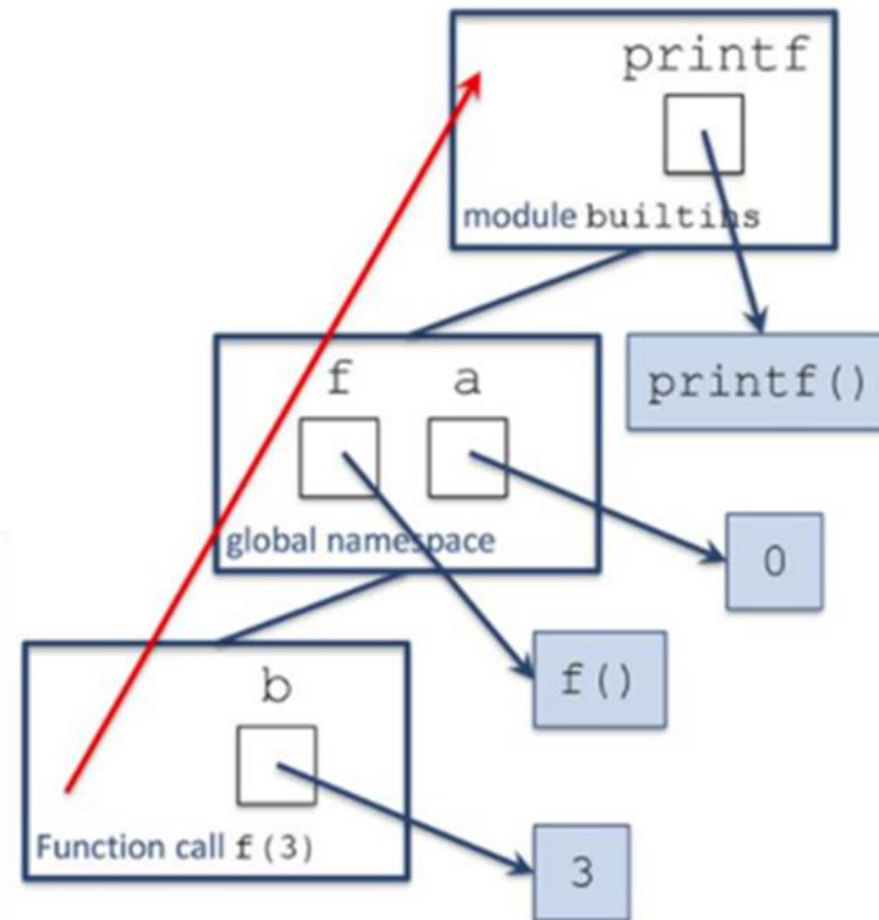
a = 0              # this a has global scope
print('f(3) = ', f(3))
print('a = ', a)   # global a is still 0
```



```
>>> === RESTART ===
>>>
f(3) = 0
a = 0
>>>
```

How Python evaluates names?

- When there are duplicate uses of a name, how does Python decide which instance of the name (e.g., local or global) you are referring to?
- To find the value of a name, Python search through namespaces in this order:
 1. First, the local (function) namespace
 2. Then the global namespace
 3. Finally, the namespace of module **builtins**



```
def f(b):          # f has global scope, b has local scope
    return a*b     # this a is the global a

a = 0              # this a has global scope
print('f(3) = ', f(3))
print('a = ', a)   # global a is still 0
```

Use dir() to show attributes

namespace4.py

```
def f(b):  
    a = 6  
    print(dir())  
    print(f.__dict__)  
    return a*b
```

```
a = 0  
print(dir())  
print('f(3)= ', f(3))  
print('a=', a)
```

```
['__annotations__', '__builtins__', '__cached__', '__doc__',  
 '__file__', '__loader__', '__name__', '__package__', '__spec__',  
 'a', 'f']
```

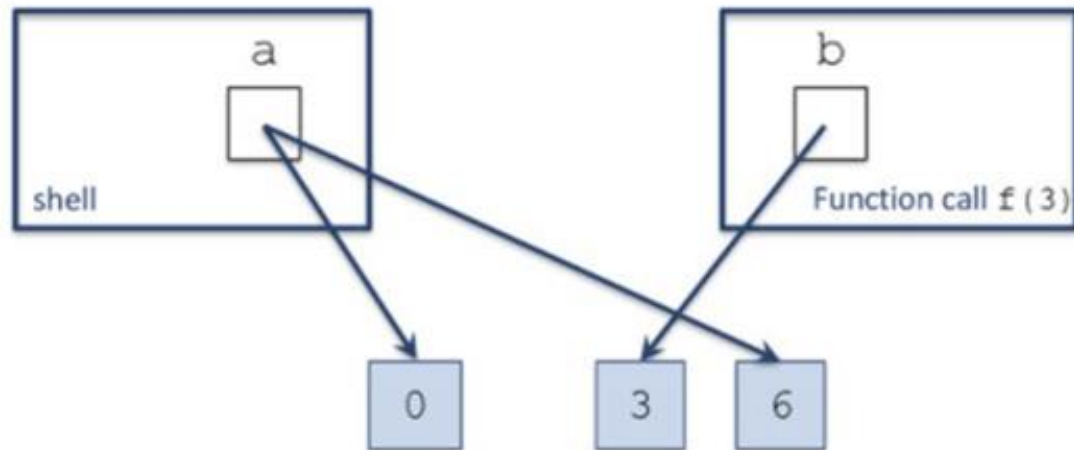
```
['a', 'b']
```

```
{  
f(3)= 18  
a= 0
```

Modifying a global variable inside a function

- To modify a global variable inside a function, use the keyword 'global'

```
def f(b):  
    global a      # all references to a in f() are to the global a  
    a = 6         # global a is changed  
    return a*b    # this a is the global a  
  
a = 0             # this a has global scope  
print('f(3) = ', f(3))  
print('a = ', a) # global a has been changed to 6
```



```
>>> === RESTART ===  
>>>  
f(3) = 18  
a = 6  
>>>
```


Lifetime and Storage Management

SECTION 3

Lifetime and Storage Management

Key events in memory management:

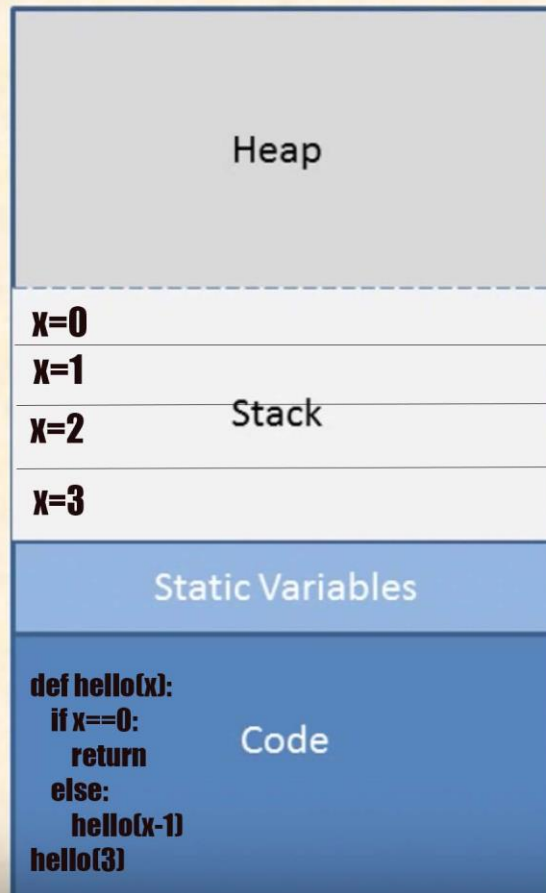
- Creation and destruction of objects
- Creation and destruction of bindings
- Deactivation and reactivation of bindings that may temporarily be unavailable
- References to variables, Subroutines, Types and so on, all of which use bindings.

Lifetime and Storage Management

- The period of time from creation to destruction is called the **Lifetime** of a binding
 - If object outlives binding it's garbage
 - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is **active** is its scope
- In addition to talking about the **scope of a binding**, we sometimes use the word **scope** as a noun all by itself, without an indirect object

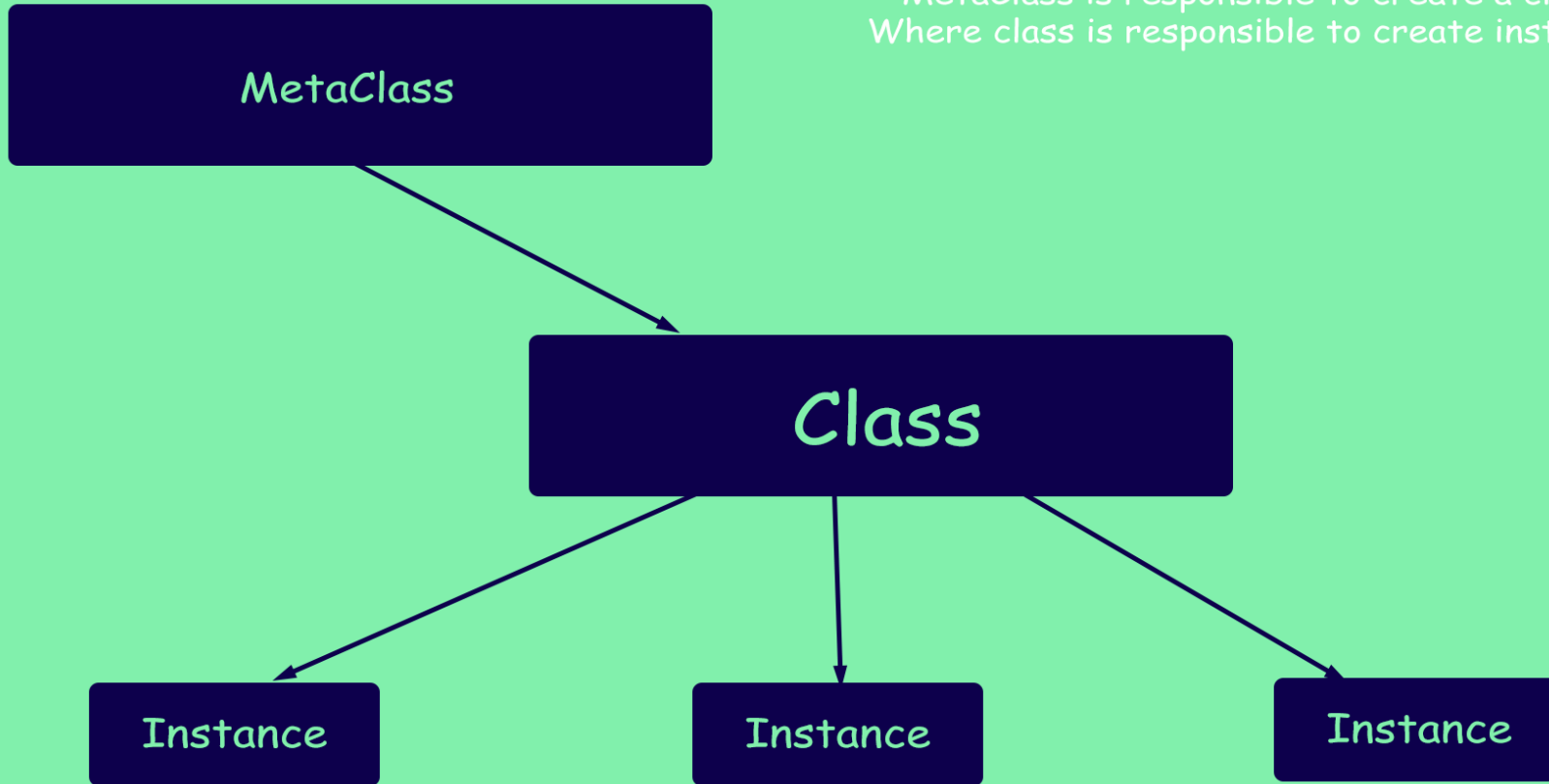
Memory

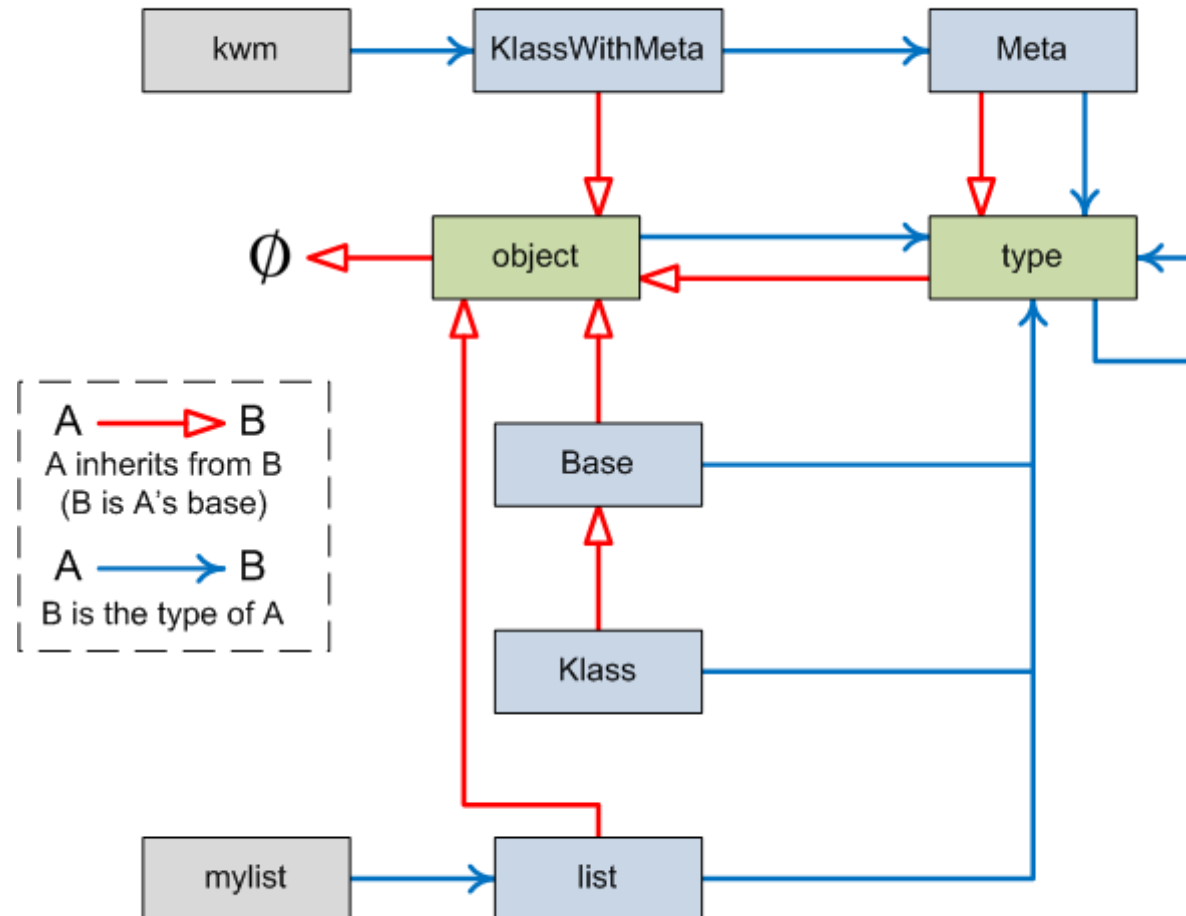
```
def hello(x):  
    if x==0:  
        return  
    else:  
        hello(x-1)  
hello(3)
```



stack will store variables or objects and it will call each variable by top of stack while returning because its works on last in first out .

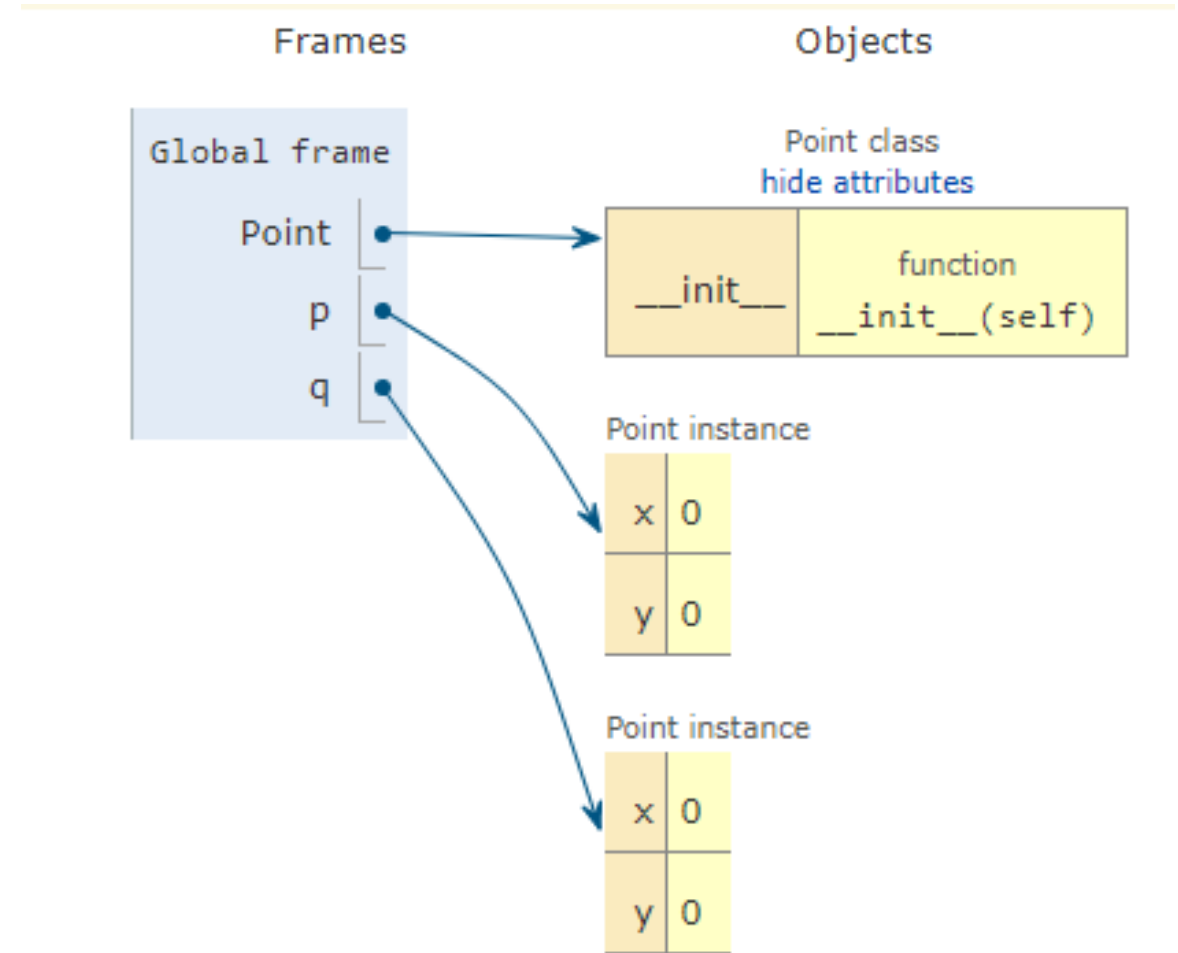
MetaClass is responsible to create a class
Where class is responsible to create instances





```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
p = Point(0, 0)
q = Point(0, 0)
```



Lifetime and Storage Management

- Contents of a stack frame
 - arguments and returns
 - local variables
 - temporaries
 - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

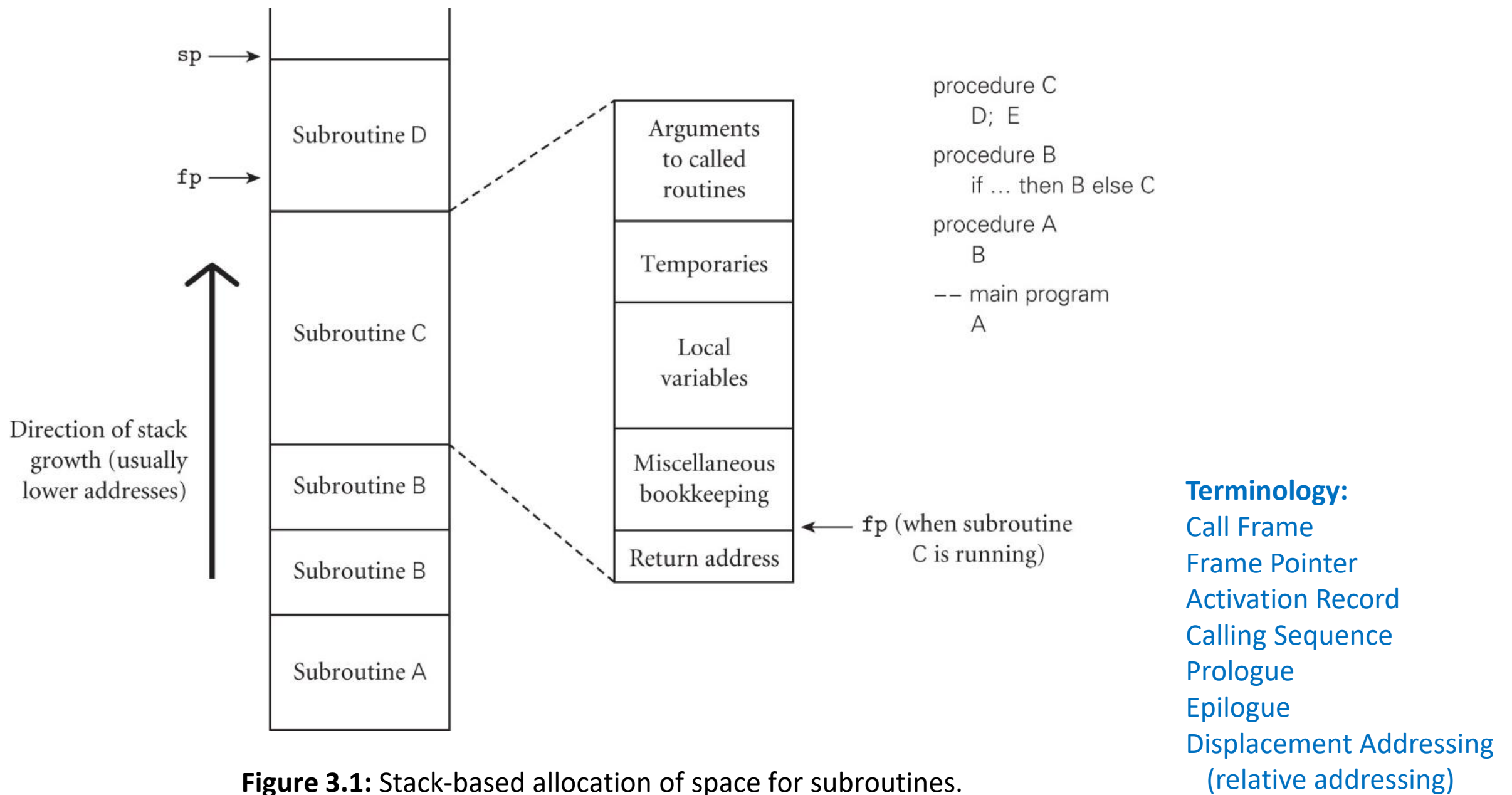


Figure 3.1: Stack-based allocation of space for subroutines.

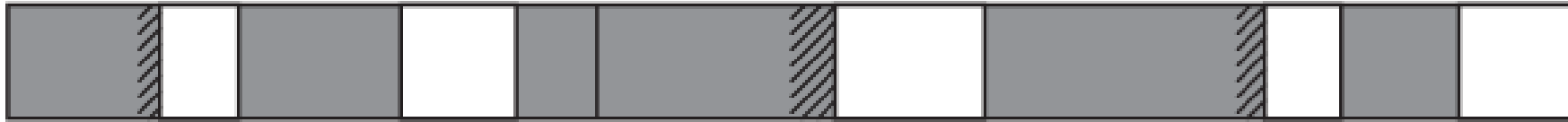
Stack and Call Frame

Maintenance of stack is responsibility of calling sequence and subroutine **prologue** and **epilogue**

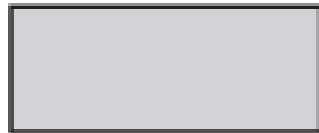
- space is saved by putting as much in the prologue and epilogue as possible
- time may be saved by putting more in the body of the subroutine
 - putting stuff in the caller instead
or
 - combining what's known in both places
(inter-procedural optimization)

Heap for dynamic allocation

Heap



Allocation request



Terminology:

Constructor

Destructor

malloc()

free()

Heap

Fragmentation

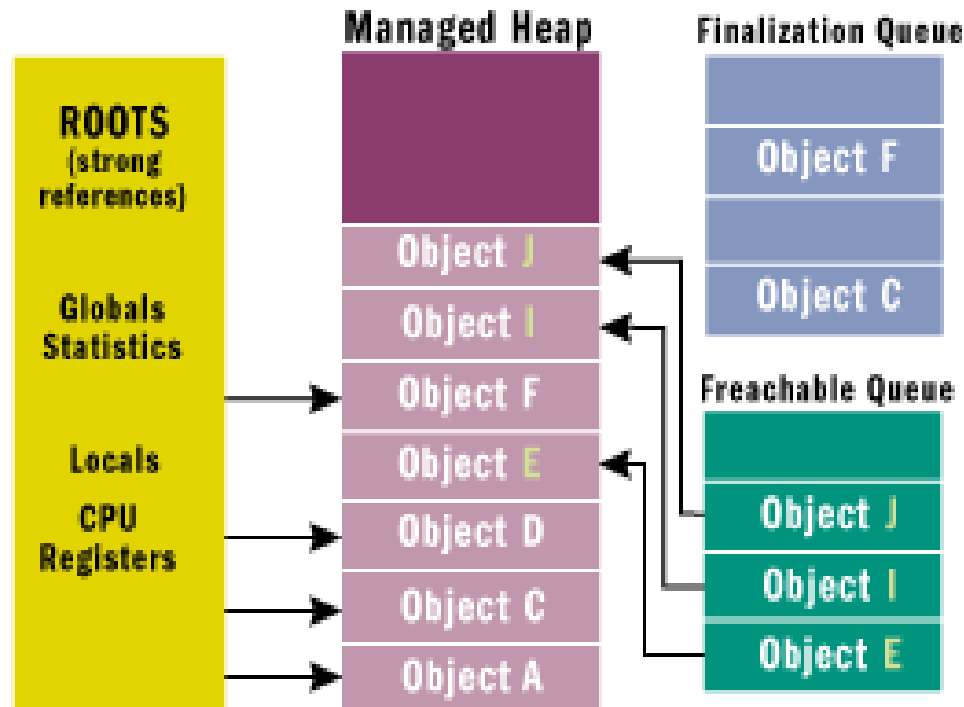
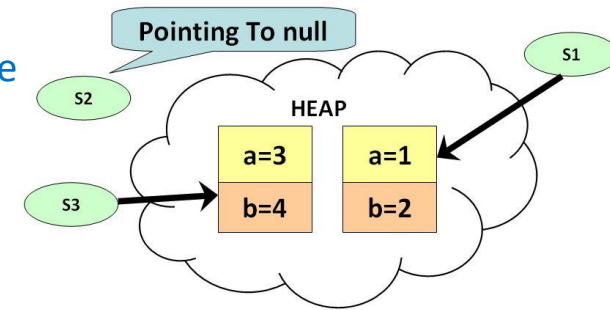
First-fit algorithm

Best-fit algorithm

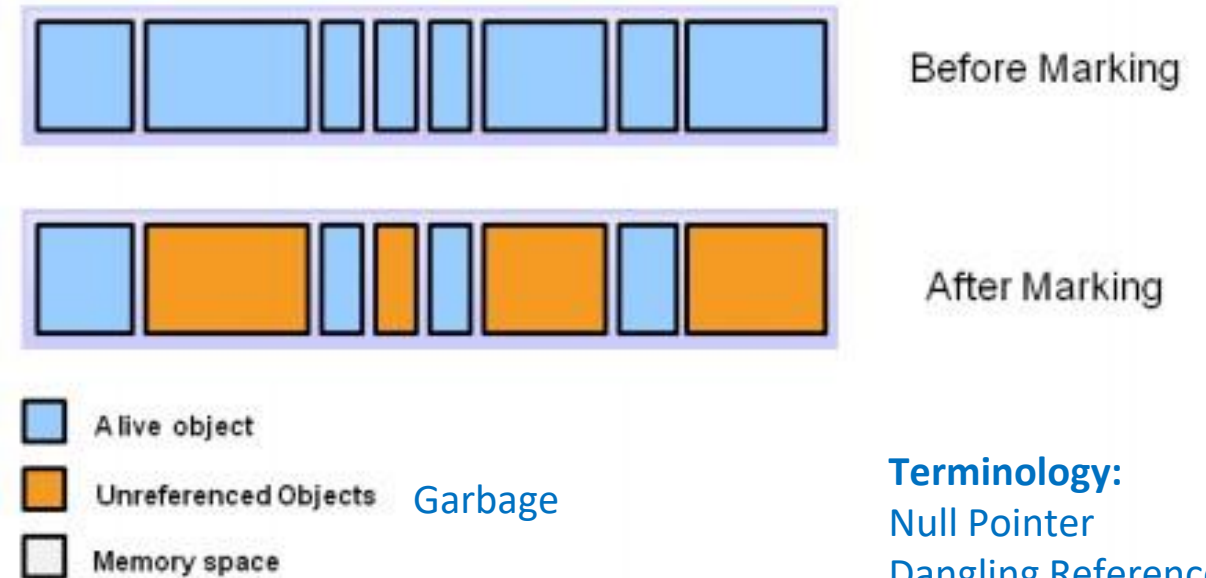
Figure 3.2 Fragmentation. The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontinuous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Garbage Collection

Dangling Reference



Marking Mark and Sweep (Chapter 8)



Terminology:
Null Pointer
Dangling Reference
Garbage

Scope Rules

SECTION 4

Scope Rules

- The textual region of the program in which a binding is active is its **scope** (of binding).
- In most languages with subroutines, we open a new scope on subroutine entry:
 - create bindings for new local variables,
 - deactivate bindings for global objects that are hidden by local object of the same name. **[local comes first in binding]**
 - re-declared make references to variables

Static vs. Dynamic Scope

Static scoping

- Local understanding of function behavior
- Know at compile-time what each name refers to
- A bit trickier to implement

Dynamic scoping

- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (just keep a global table of stacks of variable/value bindings)

Scope Rules

- **Reference Environment:** The set of active **bindings** is called the current referencing environment.
 - The set is principally determined by static or dynamic scope rules.
 - A referencing environment generally corresponds to a sequence of scopes that can be examined to find the current binding for a given name.

Scope Rules

- Referencing environments also depend on what are called **Binding Rules**.
 - Specifically, when a reference to a subroutine *S* is stored in a variable, passed as a parameter to another subroutine, or returned as a function value, one needs to determine when the referencing environment for *S* is chosen—that is, when the binding between the reference to *S* and the referencing environment of *S* is made.
 - The two principal options are:
 - **deep binding**, in which the choice is made when the reference is first created.
 - **shallow binding**, in which the choice is made when the reference is finally used.

Note: Binding and Linking are different. Binding is to associate a variable to a memory location. Linking is to associate a set of program implementation to a certain caller.

Scope Rules

- With **Static (Lexical) Scope Rules**, a scope is defined in terms of the physical (lexical) structure of the program
 - The determination of scopes can be made by the compiler
 - All bindings for identifiers can be resolved by examining the program
 - Typically, we choose the most recent, **active binding** made at compile time
 - Most compiled languages, C and Pascal included, employ static scope rules

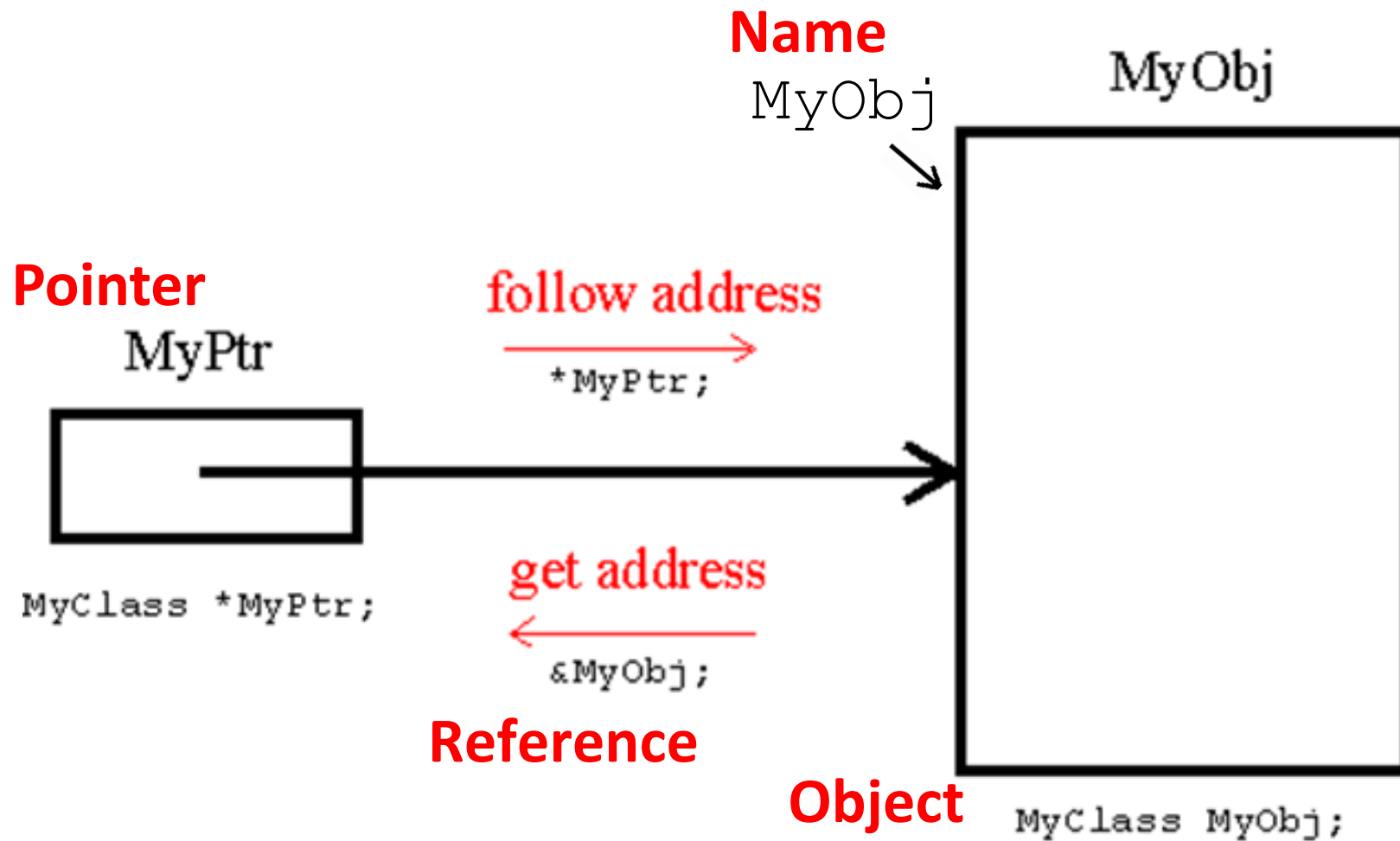
Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
 - early Lisp dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

The Meaning of Name Within a Scope

SECTION 5

A Pointer holds the address of an object



A *name* is a mnemonic character string representing something else:

- x, sin, f, prog1, null? are names
- 1, 2, 3, "test" are not names
- +, <=, ... may be names if they are not built-in operators

A *binding* is an association between two entities:

- Name and memory location (for variables)
- Name and function

Typically a binding is between a name and the object it refers to.

A *referencing environment* is a complete set of bindings active at a certain point in a program.

The *scope of a binding* is the region of a program or time interval(s) in the program's execution during which the binding is active.

A *scope* is a maximal region of the program where no bindings are destroyed (e.g., body of a procedure).

The Meaning of Names within a Scope

- Two or more names that refer to the same object at the same point in the program are said to be **aliases**.
- A name that can refer to more than one object at a given point in the program is said to be **overloaded**.
- Redefinition of a name is **overriding**.
- **Overloading** is related to the more general subject of **polymorphism**, which allows a subroutine or other program fragment to behave in different ways depending on the types of its arguments.

Aliasing

- What are aliases good for?
 - space saving - modern data allocation methods are better
 - multiple representations – unions/variants are better
 - Pointer-based data structure - linked data structures – different way of traversal.
 - Call by reference in C++ (Textbook 4e missing a statement)
 - Python does not support call by reference or creating alias for an object

Overloading

- some overloading happens in almost all languages
 - integer + vs. real +
 - read and write in Pascal
 - function return in Pascal
- some languages get into overloading in a big way
 - Ada
 - C++

Overloading and Overriding functions

- overloaded functions - two different things with the same name;
- polymorphic functions -- one thing that works in more than one way. In Python, it is named as duck functions

Environment

SECTION 5

Python Environment:

- What is Reference Environment
- Multiple environments
- Environments for HOFs
- Local names
- Function composition
- Self-referencing functions
- Currying

Binding of Referencing Environments

- The **Referencing Environment** of a statement is the collection of all names that are visible in the statement.
- **Static scope rules** specify that the referencing environment depends on the lexical nesting of program blocks in which names are declared.
- **Dynamic scope rules** specify that the referencing environment depends on the order in which declarations are encountered at run time.

Deep and Shallow Binding

- Procedure **print_selected_records** is a general-purpose routine that knows how to traverse the records in a database. It takes as parameters a database, a predicate to make print/don't print decisions, and a subroutine that knows how to format the data in the records of this database.
- Here we have hypothesized that **print_person** uses the value of nonlocal variable `line` length to calculate the number and width of columns in its output.
- In a language with dynamic scoping, it is natural for procedure **print_selected_records** to declare and initialize this variable locally, knowing that code inside **print_routine** will pick it up if needed. For this coding technique to work, the referencing environment of **print_routine** must not be created until the routine is actually called by **print_selected_records**.
- This late binding of the referencing environment of a subroutine that has been passed as a parameter is known as **shallow binding**. It is usually the default in languages with **dynamic scoping**.

Note: dynamic scoping using reference environment at run-time. Deep binding fixes RE when function is really called.

```
type person = record
```

```
...
```

```
  age : integer
```

```
...
```

```
threshold : integer
```

```
people : database
```

```
function older_than_threshold(p : person) : boolean
```

```
  return p.age ≥ threshold
```

Adjust for the printer hardware

```
procedure print_person(p : person)
```

```
  -- Call appropriate I/O routines to print record on standard output.
```

```
  -- Make use of nonlocal variable line_length to format data in columns.
```

```
...
```

shallow binding

```
procedure print_selected_records(db : database;
```

```
  predicate, print_routine : procedure)
```

```
  line_length : integer
```

```
  if device_type(stdout) = terminal
```

```
    line_length := 80
```

```
  else    -- Standard output is a file or printer.
```

```
    line_length := 132
```

```
  foreach record r in db
```

```
    -- Iterating over these may actually be
```

```
    -- a lot more complicated than a 'for' loop.
```

```
    if predicate(r)
```

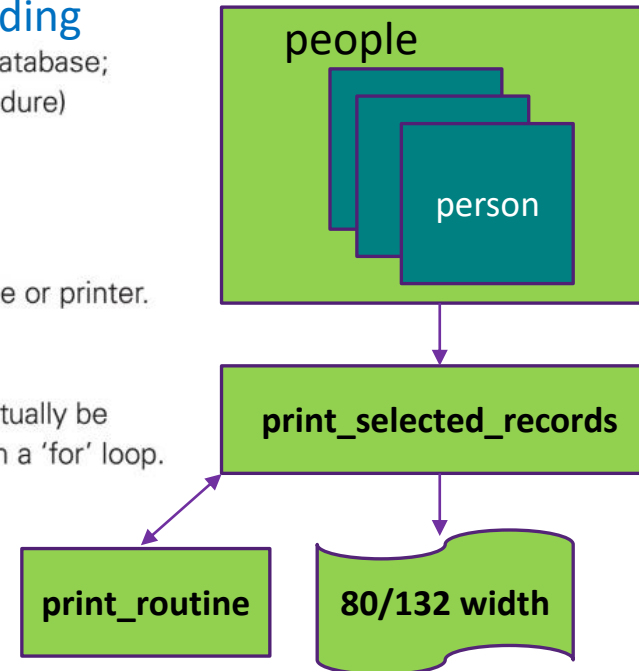
```
      print_routine(r)
```

```
-- main program
```

```
...
```

```
threshold := 35
```

```
print_selected_records(people, older_than_threshold, print_person)
```



Deep and Shallow Binding

- For function older than threshold, by contrast, shallow binding may not work well. If, for example, procedure **print_selected_records** happens to have a local variable named **threshold**, then the variable set by the main program to influence the behavior of older than threshold will not be visible when the function is finally called, and the predicate will be unlikely to work correctly.
- In such a situation, the code that originally passes the function as a parameter has a particular referencing environment in mind; it does not want the routine to be called in any other environment.
- It therefore makes sense to bind the environment at the time the routine is first passed as a parameter, and then restore that environment when the routine is finally called. This early binding of the referencing environment is known as **deep binding**.

Note: dynamic scoping using reference environment at run-time. Deep binding fixes RE when function parameter is passed.


```
type person = record
```

```
...
```

```
  age : integer
```

```
...
```

```
threshold : integer
```

```
people : database
```

Don't like global effects on the threshold.

```
function older_than_threshold(p : person) : boolean
  return p.age ≥ threshold
```

```
procedure print_person(p : person)
```

```
  -- Call appropriate I/O routines to print record on standard output.
```

```
  -- Make use of nonlocal variable line_length to format data in columns.
```

```
...
```

```
procedure print_selected_records(db : database;
```

```
  predicate, print_routine : procedure)
```

```
  line_length : integer
```

```
  if device_type(stdout) = terminal
```

```
    line_length := 80
```

```
  else    -- Standard output is a file or printer.
```

```
    line_length := 132
```

```
  foreach record r in db
```

```
    -- Iterating over these may actually be
```

```
    -- a lot more complicated than a 'for' loop.
```

```
    if predicate(r)
```

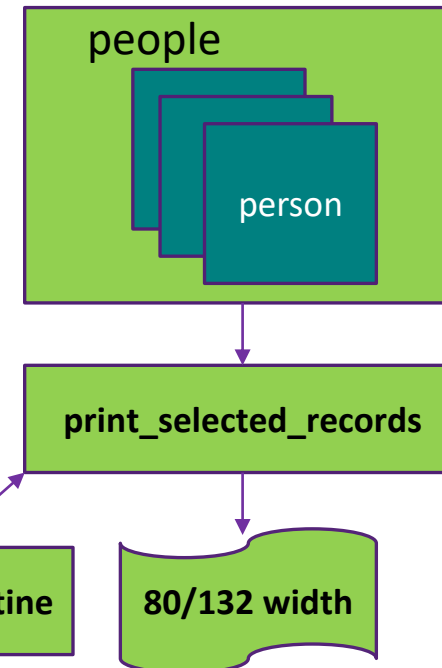
```
      print_routine(r)
```

```
-- main program
```

```
... binding fixed here.
```

```
threshold := 35
```

```
print_selected_records(people, older_than_threshold, print_person)
```



Subroutine Closures

- If your language lets you
 - (1) store subroutines as values, and
 - (2) nest subroutines, (recursive)then you can make closures.
- A **closure** is a **subroutine** that refers to variables defined in an **enclosing scope**.
- Deep binding is implemented by creating an explicit representation of a referencing environment and bundling it together with a reference to the subroutine.
- The bundle as a whole is referred to as a closure.

Subroutine Closures

- A closure in a language with static scoping capture the current instance of every object, at the time the closure is created. When the closure's subroutine is called, it will find these captured instances, even if newer instances have subsequently been created by recursive calls.
- Static scoping (by declaration of functions) with **shallow binding** is imaginable.
- Python program using static scoping but deep binding because of the interpreter-based nature.

Deep Binding or Shallow Binding

Deep/shallow binding makes sense only when a procedure can be passed as an argument to a function.

- **Deep binding** binds the environment at the time a procedure is passed as an argument.
- **Shallow binding** binds the environment at the time a procedure is actually called.

Subroutine Closures

```
def A(I, P):  
    def B():  
        print(I)  
    # body of A:  
    if I > 1:  
        P()  
    else:  
        A(2, B)  
  
def C():  
    pass      # do nothing  
  
A(1, C)      # main program
```

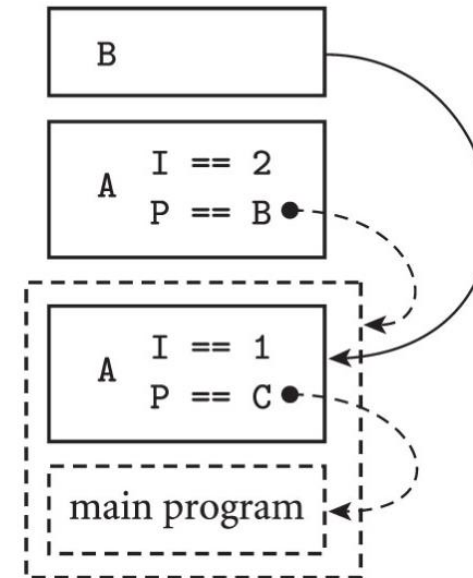
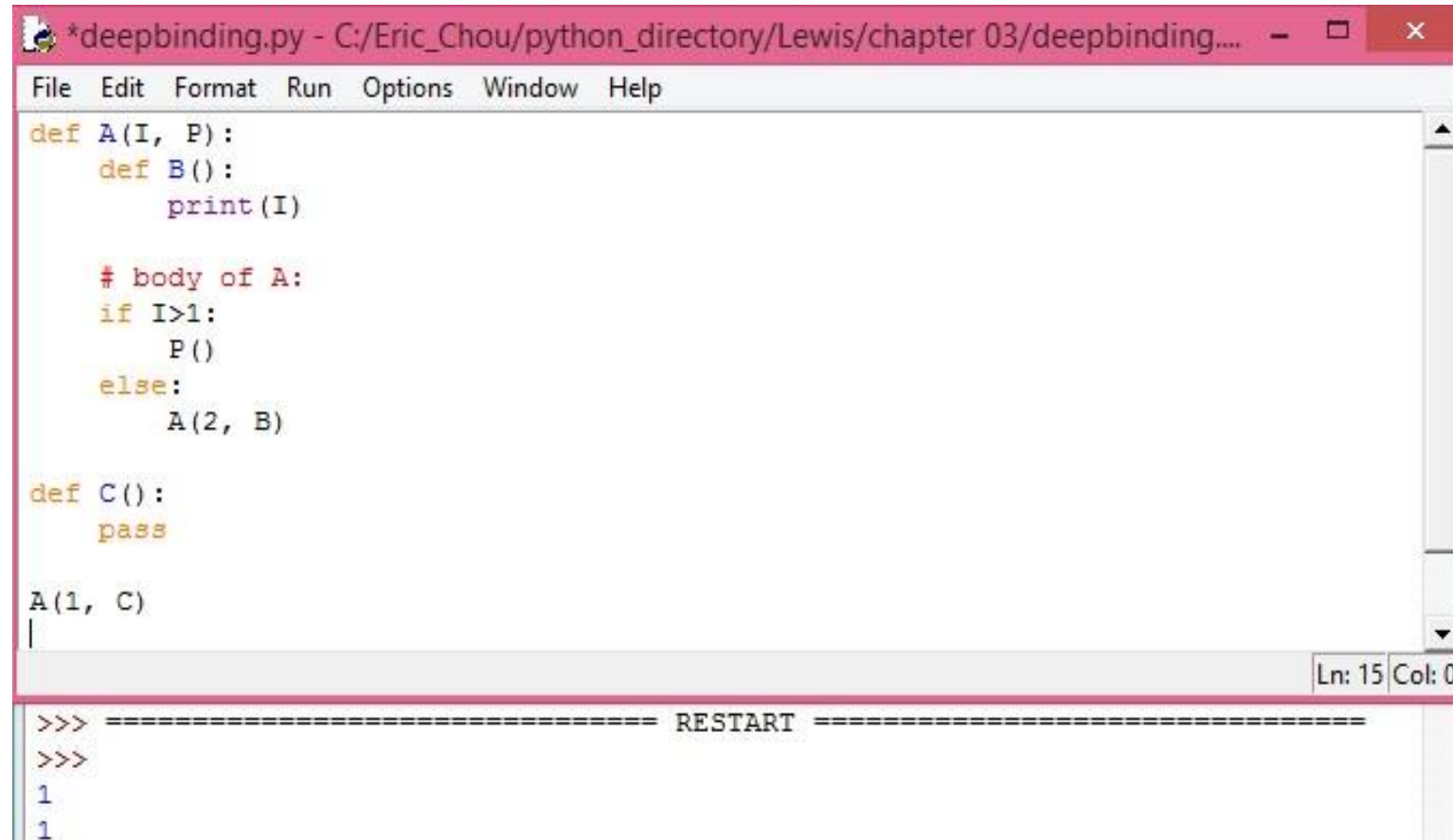


Figure 3.14 Deep binding in Python.

At right is a conceptual view of the run-time stack. Referencing environments captured in closures are shown as dashed boxes and arrows. When B is called via formal parameter P, two instances of I exist. Because the closure for P was created in the initial invocation of A, B's static link (solid arrow) points to the frame of that earlier invocation. B uses that invocation's instance of I in its print statement, and the output is a 1.

Note: With shallow binding, it should print 2.

Demo Program: deepbinding.py



```
*deepbinding.py - C:/Eric_Chou/python_directory/Lewis/chapter 03/deepbinding....
File Edit Format Run Options Window Help

def A(I, P):
    def B():
        print(I)

    # body of A:
    if I>1:
        P()
    else:
        A(2, B)

def C():
    pass

A(1, C)
|

Ln: 15 Col: 0

>>> ===== RESTART =====
>>>
1
1
```

Newer Version (3.6+)

```
def A(I, P):  
    def B():  
        print(I)  
    # body of A  
    if I>1:  
        P()  
    else:  
        A(2, B)
```

```
def C():  
    pass  
A(1, C)
```

deepbinding1.py

```
[Running] python -u "c:\Eric_Chou\Lewis University  
1|
```

```
[Done] exited with code=0 in 0.747 seconds
```

First-Class Values and Unlimited Event

- A value in a programming language is said to have **first-class** status if it can be passed as a parameter, returned from a subroutine, or assigned into a variable.
- By contrast, a “second-class” value can be passed as a parameter, but not returned from a subroutine or assigned into a variable.

[Algol]

- A “third-class” value cannot even be passed as a parameter. **[Label]**

Note: A language construct is said to be a First-Class value in that language when there are no restrictions on how it can be created and used: when the construct can be treated as a value without restrictions.

Returning a First-Class Sub-Routine in Scheme

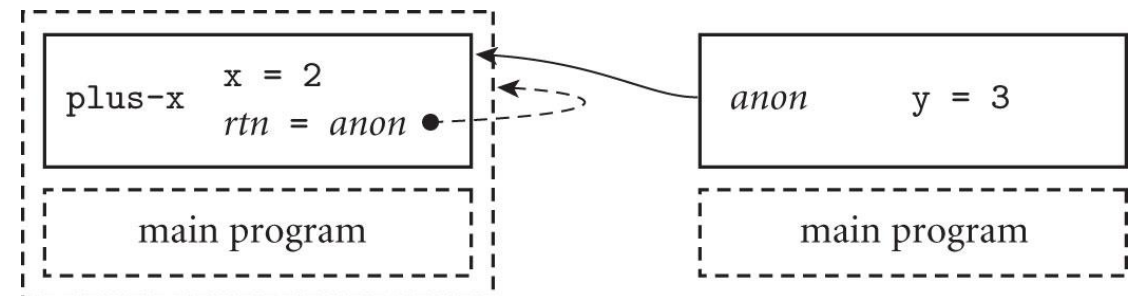
- First-class subroutines in a language with nested scopes introduce an additional level of complexity: they raise the possibility that a reference to a subroutine may outlive the execution of the scope in which that routine was declared. Consider the example in Scheme on the right:
- Here the `let` construct on line 4 declares a new function, **f**, which is the result of calling **plus-x** with argument **2**. Function **plus-x** is defined at line 1. It returns the (unnamed) function declared at line 2. But that function refers to parameter **x** of **plus-x**. When **f** is called at line 5, its referencing environment will include the **x** in **plus-x**, despite the fact that **plus-x** has already returned (see Figure 3.15). Somehow, we **must** ensure that **x** remains available.

f => **lambda** (2)(**lambda** (y)(+ 2 y))
(f 3) => **lambda**(3)(+ 2 y)

```
1. (define plus-x (lambda (x)
2.   (lambda (y) (+ x y))))
3. ...
4. (let ((f (plus-x 2)))
5.   (f 3))
```

One Reference Environment ; returns 5

```
(define plus-x (lambda (x) (lambda (y) (+ x y)
                                )
                )
              )
              B = lambda (x) (A)
              define plus-x (B)
```



Object Closures

- In object-oriented languages, there is an alternative way to achieve a similar effect: we can encapsulate our subroutine as a method of a simple object, and let the object's fields hold context for the method.

```
class Int_func:
    def __init__(self, x):
        self.x = x
    def __call__(self, y):
        return self.x + y

plus2 = Int_func(2)
print(plus2(3))
```

Int_func.py

5

Lambda Expressions

- A lambda expression is an anonymous function that you can use to create delegates or expression tree types. [Anonymous Inner Function or Anonymous Inner Class]

Python lambda construction, you get the following:

```
>>>>> lambda x: x
```

In the example above, the expression is composed of:

- **The keyword:** lambda
- **A bound variable:** x
- **A body:** x

Usage Of Lambda Expression

lambda1.py

```
print("Lambda Function as Callable: ")  
print((lambda x: x*x) (2))
```

```
cube = lambda x: x**3
```

```
print("Lambda Function as Short Function Definition: ")  
print(cube(2))
```

```
a = [(3, 1), (5, 6), (4, -1), (8, 2)]  
a.sort(key=lambda t: t[1])  
print("Lambda Function as a key: ")  
print(a)
```

Lambda Function as Callable:

4

Lambda Function as Short Function Definition:

8

Lambda Function as a key:

[(4, -1), (3, 1), (8, 2), (5, 6)]

Multiple Environments

SECTION 5.1

Life cycle of a function

What happens?

Def statement

```
def square ( x ) :  
    return x * x
```

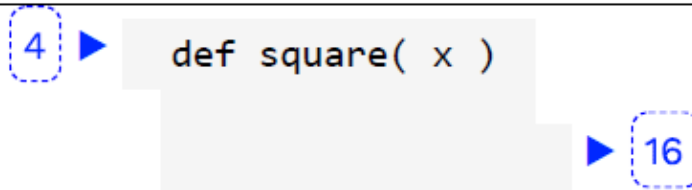
- A new function is created!
- Name bound to that function in the current frame.

Call expression

```
square ( 2 + 2 )
```

- Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)

Calling/applying



- A new frame is created!
- Parameters bound to arguments
- Body is executed in that new environment

A nested call expression

1. `def square (x) :`
2. `return x * x`
3. `square (square (3))`

A nested call expression

```
1. next def square (x) :  
2.         return x * x  
3.         square (square (3) )
```


A nested call expression

```
1. prev def square(x):  
2.         return x * x  
3. next square(square(3))
```

Global frame

square | ● ----> func square(x) [parent=Global]

```
square (square (3) )
```

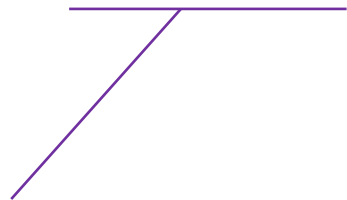
A nested call expression

```
1. prev def square(x):  
2.         return x * x  
3. next square(square(3))
```

Global frame

square | ● ----> func square(x) [parent=Global]

square (square (3))



func square (x)

A nested call expression

```
1. prev def square(x):  
2.         return x * x  
3. next square(square(3))
```

Global frame

square | ● ----> func square(x) [parent=Global]

square (square (3))

func square (x)

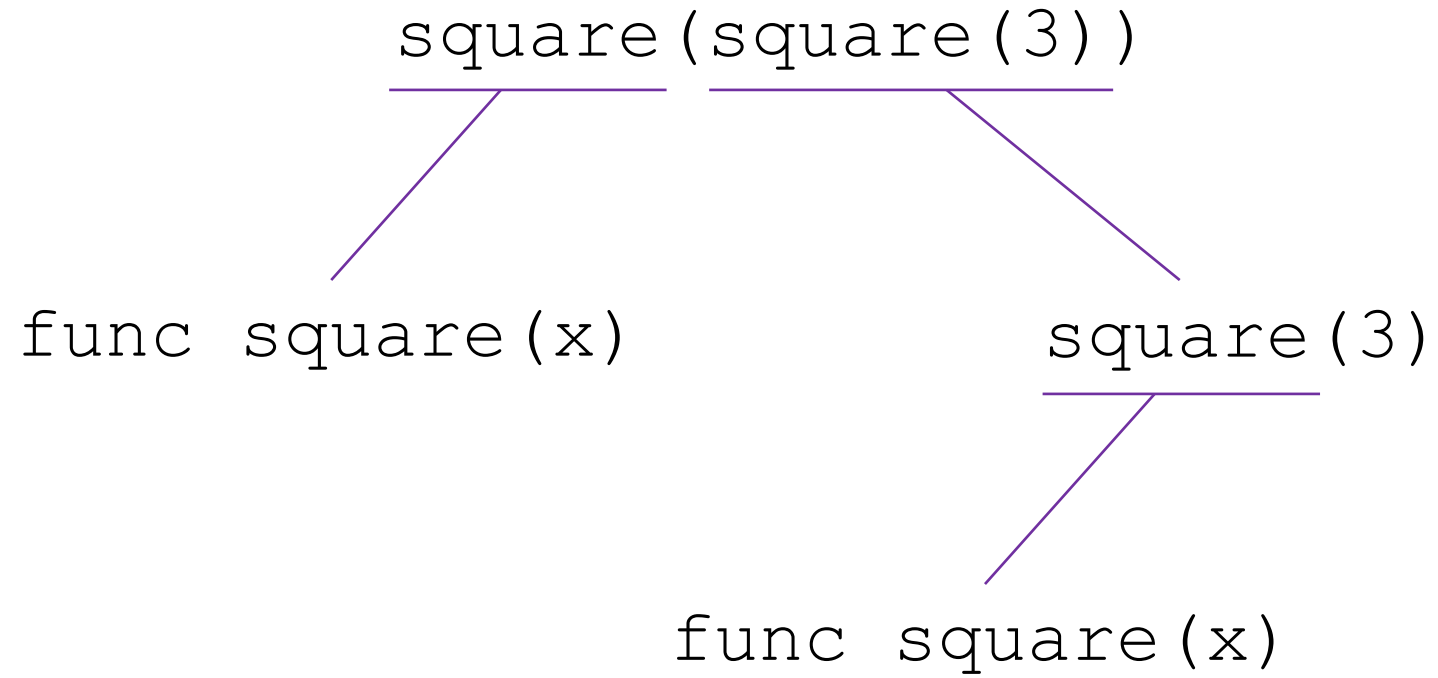
square (3)

A nested call expression

```
1. prev def square(x):  
2.         return x * x  
3. next square(square(3))
```

Global frame

square | ● ----> func square(x) [parent=Global]

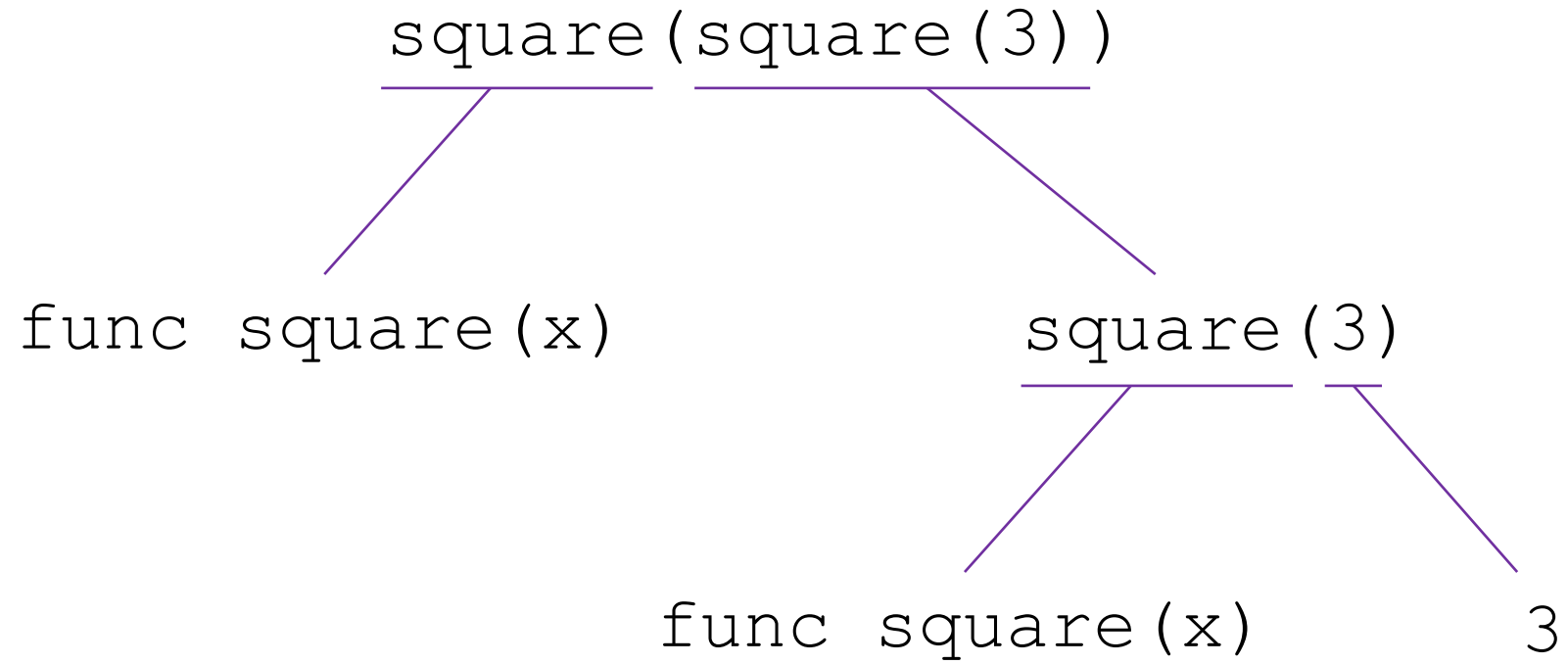


A nested call expression

```
1. prev def square(x):  
2.         return x * x  
3. next square(square(3))
```

Global frame

square | ● ----> func square(x) [parent=Global]

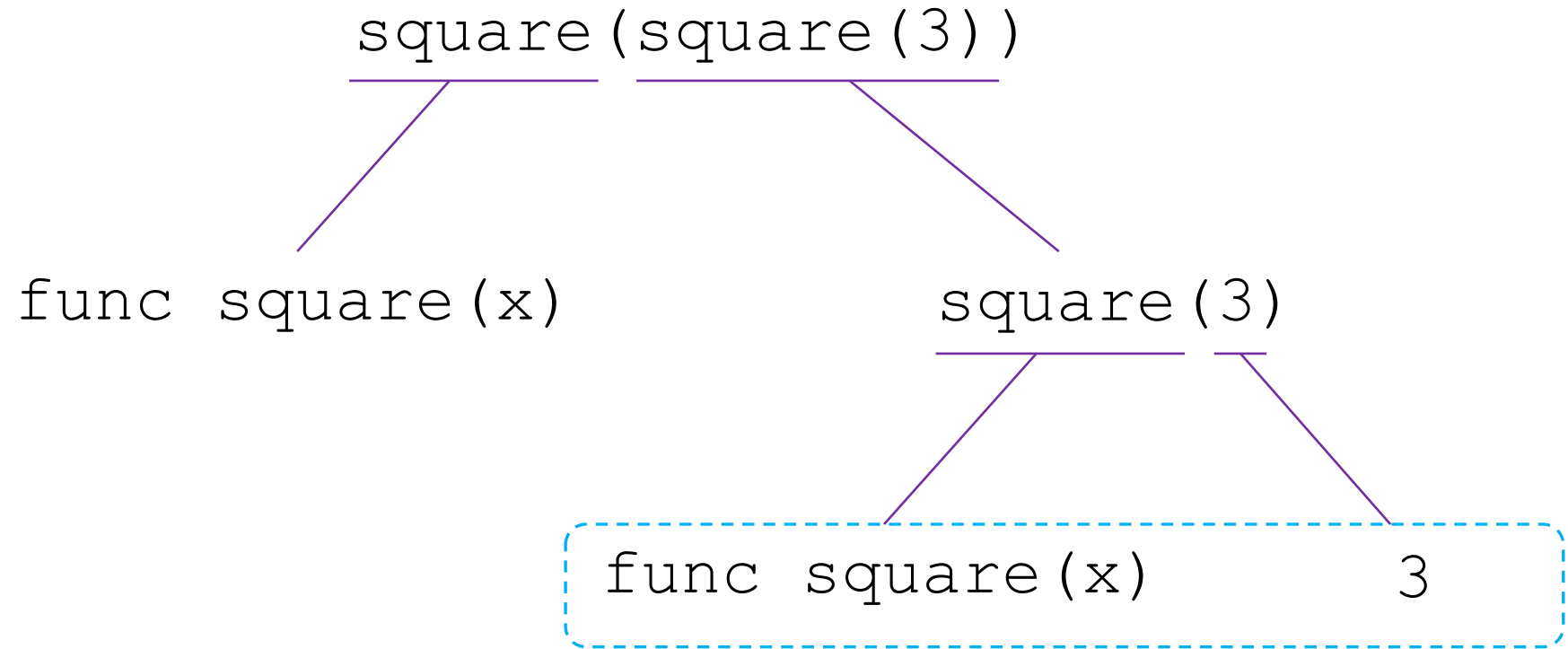


A nested call expression

```
1. prev def square(x):  
2.         return x * x  
3. next square(square(3))
```

Global frame


square | ● ----> func square(x) [parent=Global]



A nested call expression

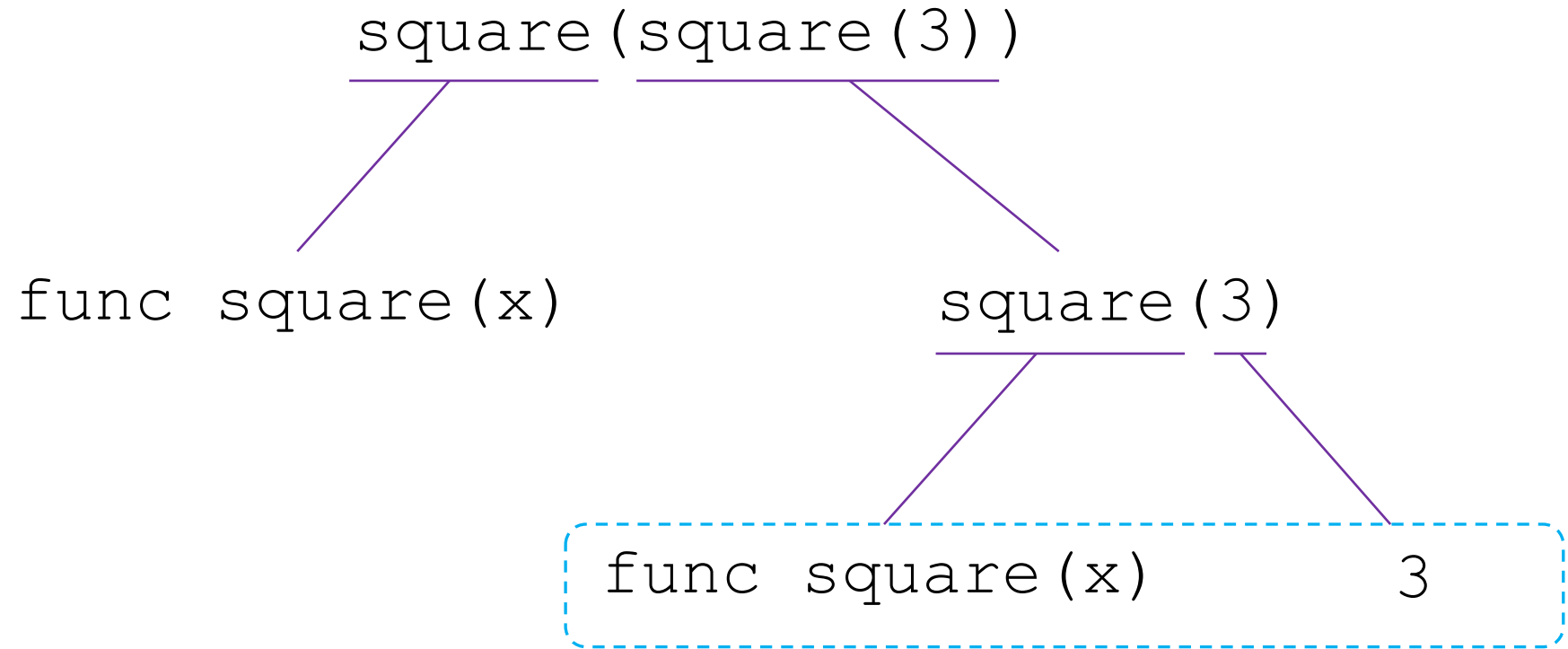
```
1. next def square(x):  
2.     return x * x  
3. prev square(square(3))
```

Global frame

square  ----> func square(x) [parent=Global]

f1: square [parent=Global]


x | 3



A nested call expression

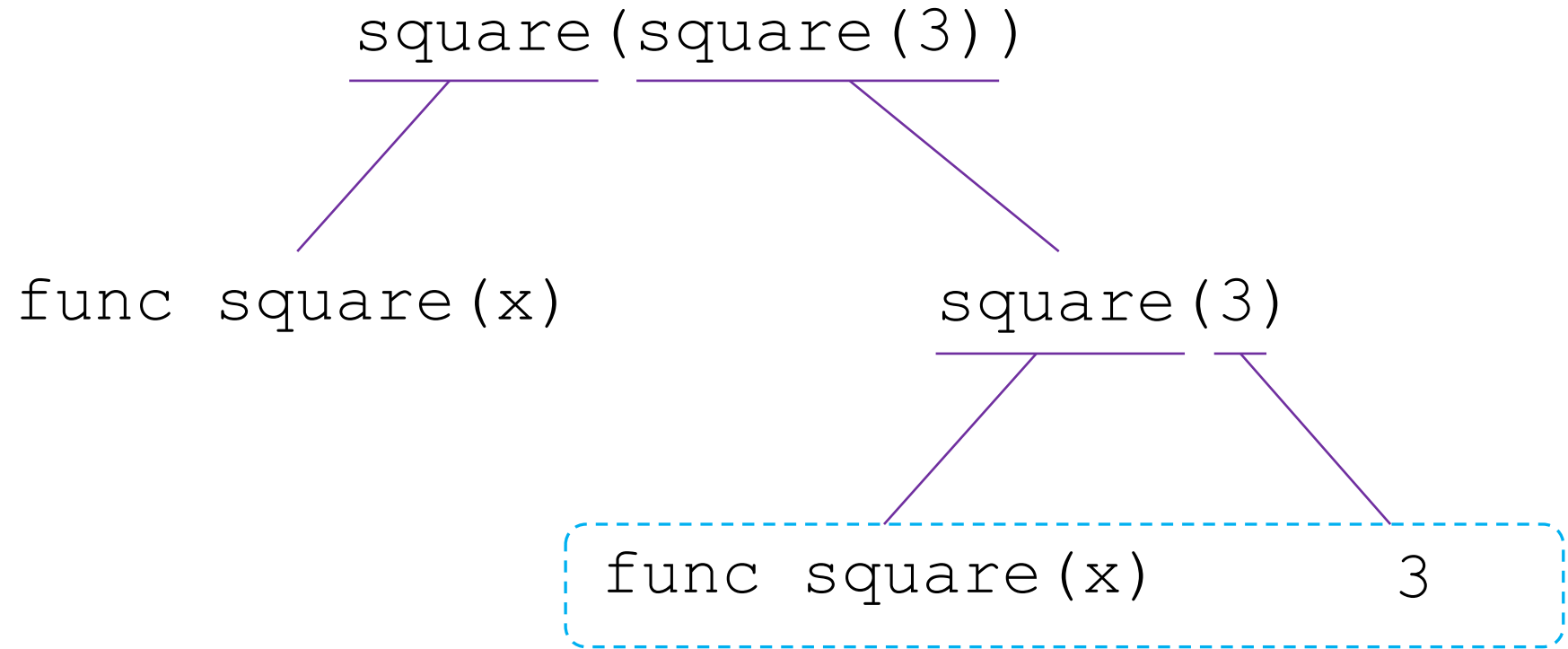
```
1. prev def square(x):  
2. next     return x * x  
3.         square(square(3))
```

Global frame

square  ----> func square(x) [parent=Global]

f1: square [parent=Global]


x | 3



A nested call expression

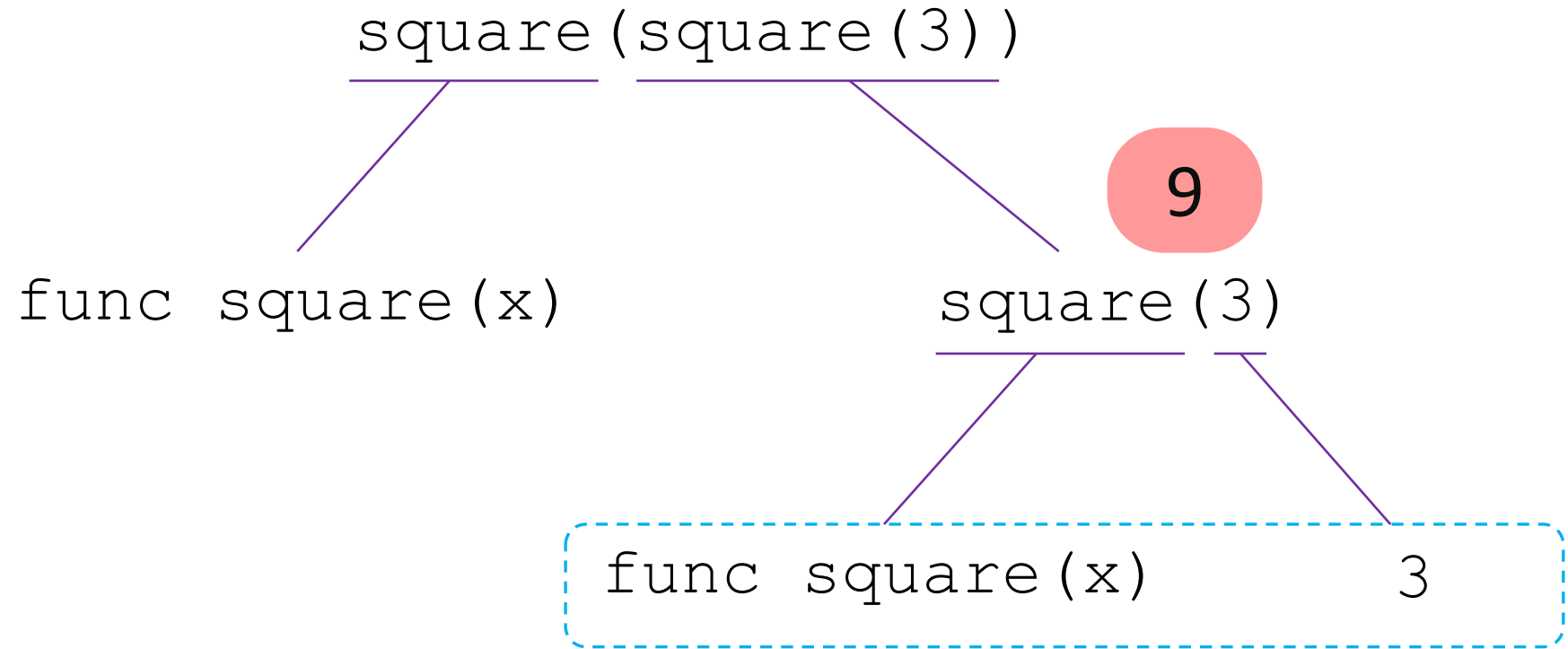
```
1. next def square(x):  
2.         return x * x  
3. prev square(square(3))
```

Global frame

square  ----> func square(x) [parent=Global]

f1: square [parent=Global]


x	3
Return value	9



A nested call expression

```
1. next def square(x):  
2.         return x * x  
3. prev square(square(3))
```

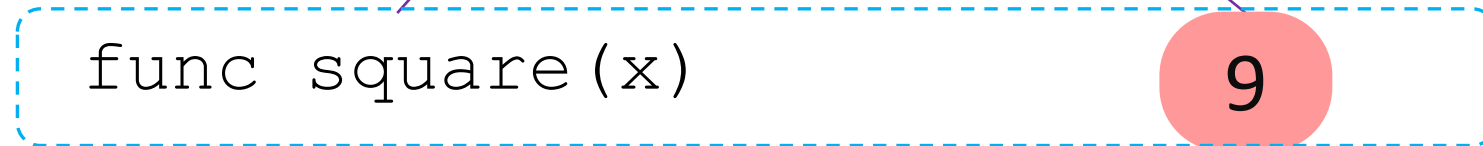
Global frame

square  ----> func square(x) [parent=Global]

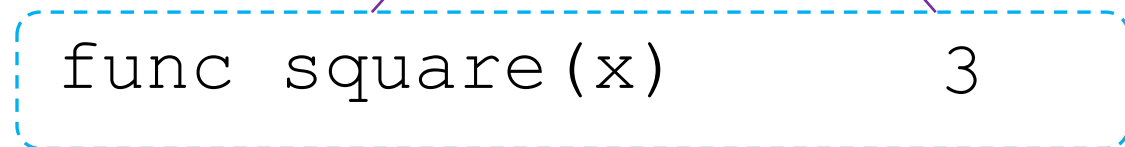
f1: square [parent=Global]

x	3
Return value	9

`square (square (3))`




`square (3)`



A nested call expression

```
1. prev def square(x):  
2. next     return x * x  
3.         square(square(3))
```

Global frame

square  ----> func square(x) [parent=Global]

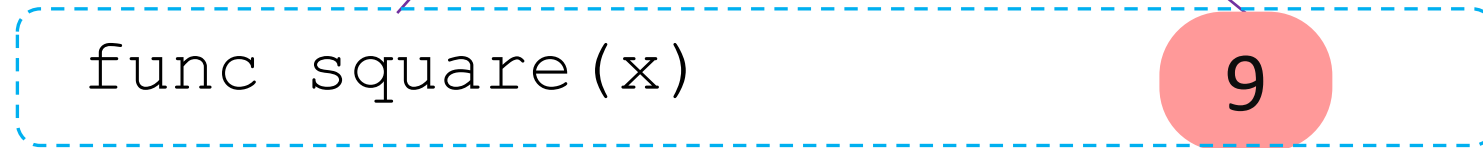
f1: square [parent=Global]

x		3
Return value		9

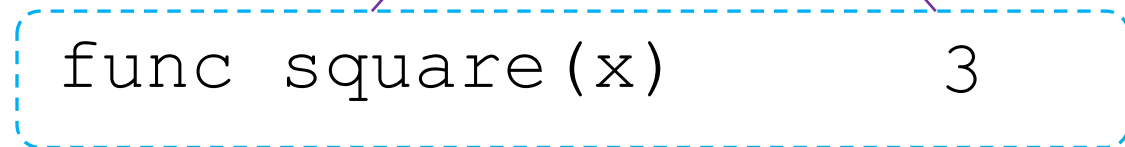
f2: square [parent=Global]

x		9
---	--	---

`square (square (3))`




`square (3)`



A nested call expression

```
1. def square(x):  
2.     return x * x  
3. square(square(3))
```

Global frame

square  ----> func square(x) [parent=Global]

f1: square [parent=Global]

x	3
Return value	9

f2: square [parent=Global]

x	9
Return value	81

81

square (square (3))

func square (x)

9

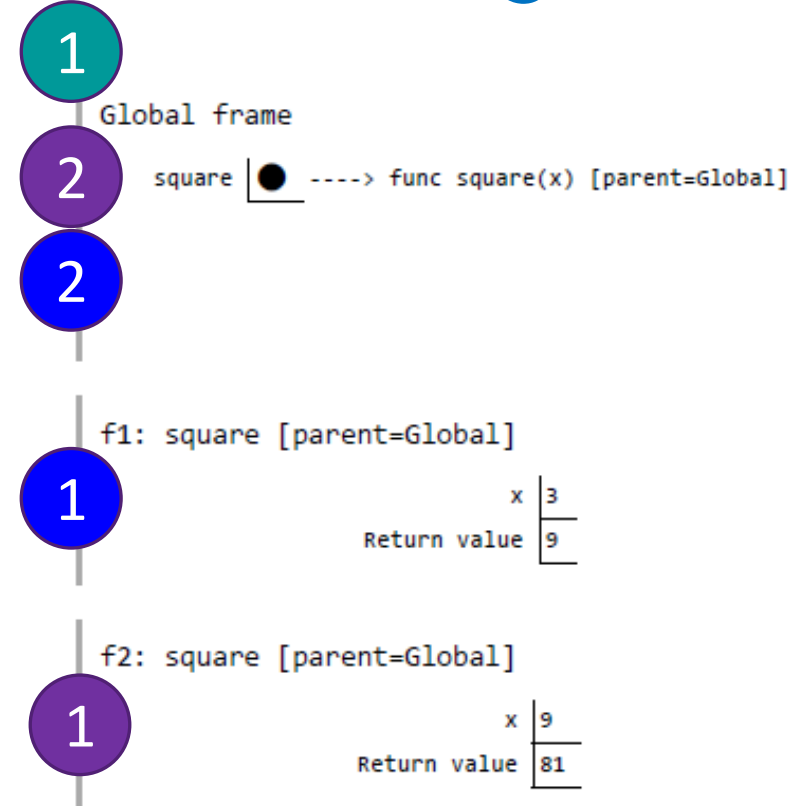
square (3)

func square (x)

3

Multiple environments in one diagram!

```
1.      def square (x) :  
2.          return x * x  
3.      square (square (3) )
```

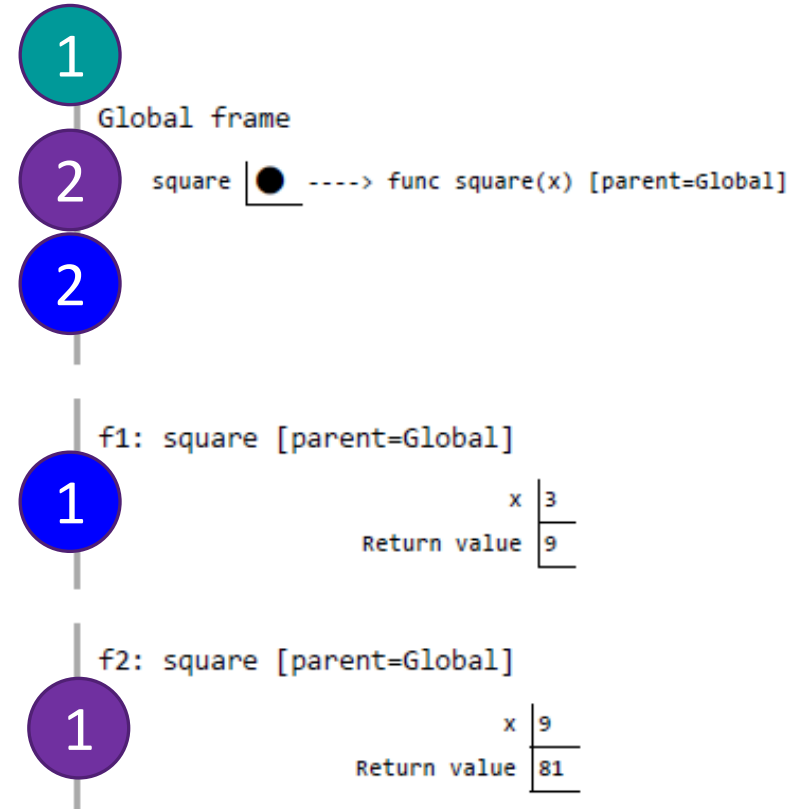


An environment is a sequence of frames.

- Environment: Global frame
- Environment: Local frame (f1), then global frame
- Environment: Local frame (f2), then global frame

Names have no meanings without environments

```
1.      def square (x) :  
2.          return x * x  
3.      square (square (3) )
```

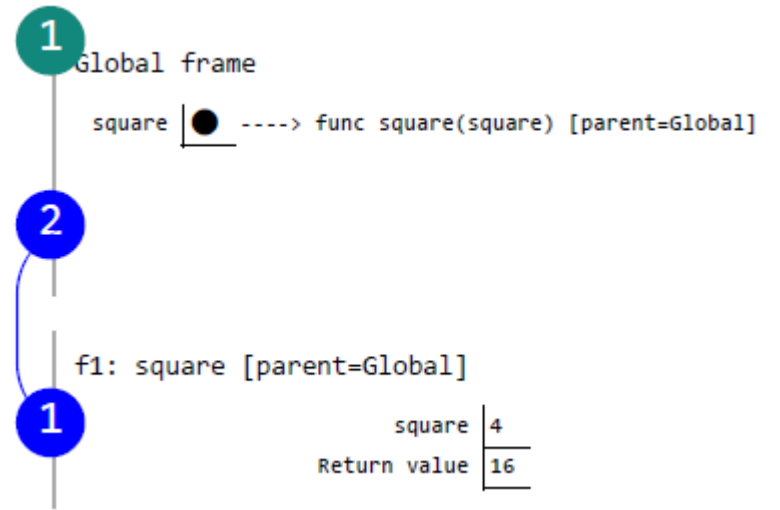


Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Names have different meanings in different environments

```
def square (square) :  
    return square * square  
  
square (4)
```



Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Environments for higher-order functions

SECTION 5.2

Review: Higher-order functions

- A higher-order function is either...
- A function that takes a function as an argument value
`summation(5, lambda x: x**2)`
- A function that returns a function as a return value
`make_adder(3)(1)`
- Functions are first class: Functions are values in Python.

Example: Apply twice

```
def apply_twice(f, x):  
    return f(f(x))
```

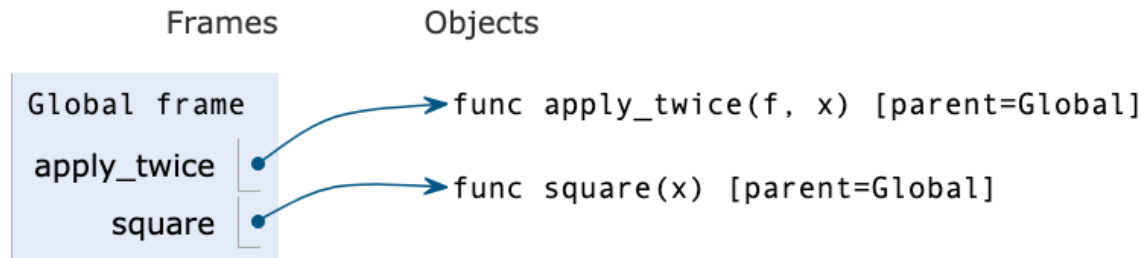
```
def square(x):  
    return x ** 2
```

```
apply_twice(square, 3)
```

Arguments bound to functions

Python 3.6
([known limitations](#))

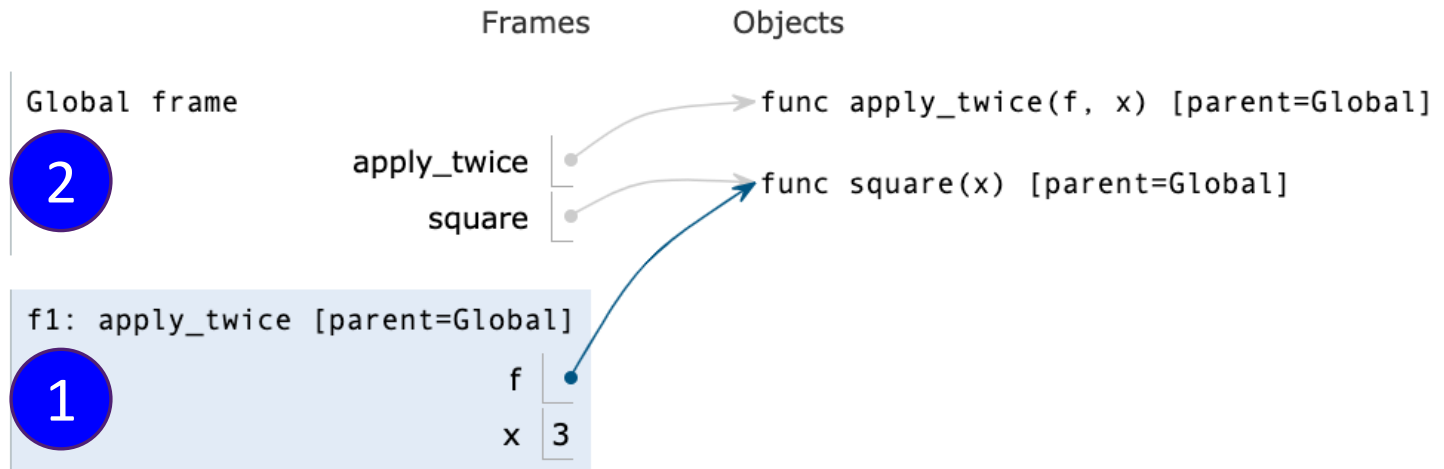
```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x ** 2  
6  
→ 7 apply_twice(square, 3)
```



Python 3.6
([known limitations](#))

```
→ 1 def apply_twice(f, x):  
2     return f(f(x))  
3  
4 def square(x):  
5     return x ** 2  
6  
→ 7 apply_twice(square, 3)
```

[Edit this code](#)



Higher Order Function (Function in a function)

higher_order1.py

```
def mul(x):  
    def g(y):  
        return x*y  
    return g  
  
print(mul(2)(3))
```

6

Higher Order Function (HOF)

higher_order2.py

```
def add(x, y):  
    return x + y  
  
def sub(x, y):  
    return x - y  
  
def go(x, y, f):  
    return f(x, y)  
  
print(go(2, 3, add))  
print(go(2, 3, sub))  
print(go(2, 3, lambda a, b: a*b))
```

5
-1
6

Environments for nested definitions

SECTION 5.3

Example: Make texter

```
def make_texter(emoji):  
    def texter(text):  
        return emoji + text + emoji  
    return texter
```

```
happy_text = make_texter("😊")  
result = happy_text("lets go to the beach!")
```

Environments for nested def statements

Python 3.6
([known limitations](#))

```
1 def make_texter(emoji):
2     def texter(text):
3         return emoji + text + emoji
4     return texter
5
6 happy_text = make_texter("😄")
7 result = happy_text("dance party!")
```

[Edit this code](#)

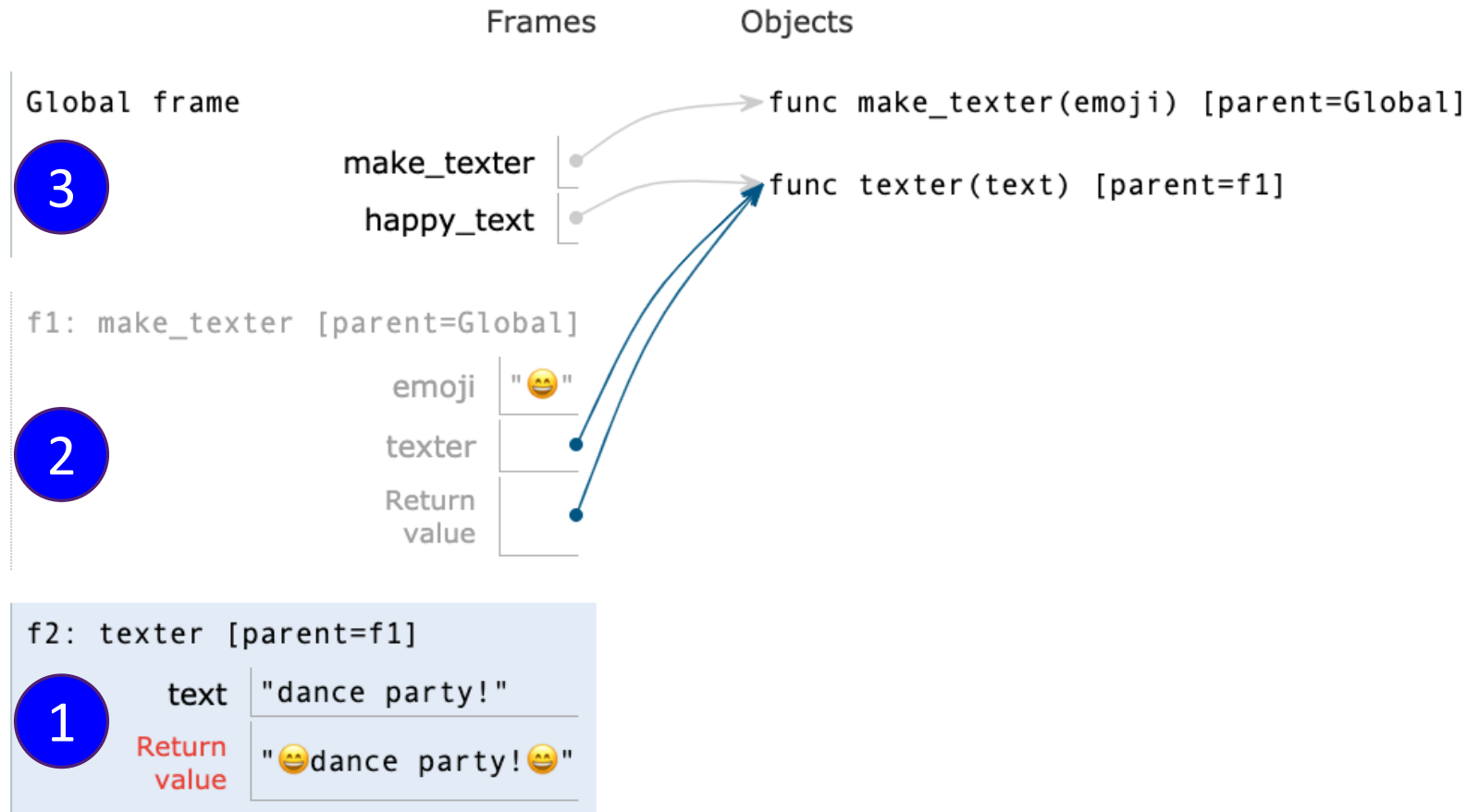
➡ line that just executed

➡ next line to execute



Step 10 of 10

[Customize visualization](#)



Environments for nested def statements

- Every user-defined **function** has a parent frame
- The parent of a **function** is the frame in which it was defined
- Every local **frame** has a parent frame
- The parent of a **frame** is the parent of the called function
- An environment is a **sequence of frames**.

How to draw an environment diagram

When a function is defined:

1. Create a function value:

`func <name>(<formal parameters>) [parent=<label>]`

2. Its parent is the current frame.
3. Bind `<name>` to the function value in the current frame

How to draw an environment diagram

When a function is called:

1. Add a local frame, titled with the `<name>` of the function being called.
2. Copy the parent of the function to the local frame:
`[parent=>label<]`
3. Bind the `<formal parameters>` to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

Local names

SECTION 5.4

Example: Thingy Bobber

```
def thingy(x, y):  
    return bobber(y)
```

localname1.py

```
def bobber(a):  
    return a + y
```

```
result = thingy("ma", "jig")
```

🤔 What do you think will happen?

Local name visibility

Local names are not visible to other (non-nested) functions.

Python 3.6
([known limitations](#))

```
1 def thingy(x, y):  
2     return bobber(x)  
3  
4 def bobber(a):  
→ 5     return a + y  
6  
7 result = thingy("ma", "jig")
```

[Edit this code](#)

→ line that just executed

→ next line to execute

Frames

Objects

Global frame

2

thingy
bobber

func thingy(x, y) [parent=Global]

func bobber(a) [parent=Global]

f1: thingy [parent=Global]

x "ma"
y "jig"

f2: bobber [parent=Global]

1

a "ma"

Local name visibility

- An environment is a sequence of frames.
- The environment created by calling a top-level function consists of one local frame followed by the global frame.

Function Composition

SECTION 5.5

Example: Composer

```
def happy(text):  
    return "😊" + text + "😊"  
  
def sad(text):  
    return "😞" + text + "😞"  
  
def composer(f, g):  
    def composed(x):  
        return f(g(x))  
    return composed  
  
msg1 = composer(sad, happy)("cs61a!")  
msg2 = composer(happy, sad)("eecs16a!")
```

 What do you think will happen?

Passing function for composition

compose.py

```
def happy(text):  
    return ":)" + text + ":)"  
  
def sad(text):  
    return ":(" + text + ":("  
  
def composer(f, g):  
    def composed(x):  
        return f(g(x))  
    return composed  
  
msg1 = composer(sad, happy)("cs61a!")  
msg2 = composer(happy, sad)("eecs16a!")  
print(msg1)  
print(msg2)
```

```
:(:)cs61a!:(  
:):(eecs16a!:(:
```

Example: Composer (Part 2)

One of the composed functions could itself be an HOF...

```
def happy(text):  
    return "😊" + text + "😊"  
  
def sad(text):  
    return "😞" + text + "😞"  
  
def make_texter(emoji):  
    def _texter(text):  
        return emoji + text + emoji  
    return _texter  
  
def composer(f, g):  
    def composed(x):  
        return f(g(x))  
    return composed  
  
composer(happy, make_texter("🌨️"))('snow day!')
```

Configurable Function and Composition of Functions

```
def happy(text):  
    return ":)" + text + ":)"  
def sad(text):  
    return ":(" + text + ":("  
def make_texter(emoji):  
    def texter(text):  
        return emoji + text + emoji  
    return texter  
def composer(f, g):  
    def composed(x):  
        return f(g(x))  
    return composed  
  
msg1 = composer(happy, make_texter("&"))(" Snow Day! ")  
print(msg1)
```

compose2.py
:)& Snow Day! &:)

Self-reference

SECTION 5.6

A self-referencing function

A higher-order function could return a function that references its own name.

```
def print_sums (n) :  
    print (n)  
    def next_sum (k) :  
        return print_sums (n + k)  
    return next_sum
```

```
print_sums (1) (3) (5)
```

Environment for print_sums

```
1 def print_sums(n):  
2     print(n)  
3     def next_sum(k):  
4         return print_sums(n + k)  
5     return next_sum  
6  
7 print_sums(1)(3)(5)
```

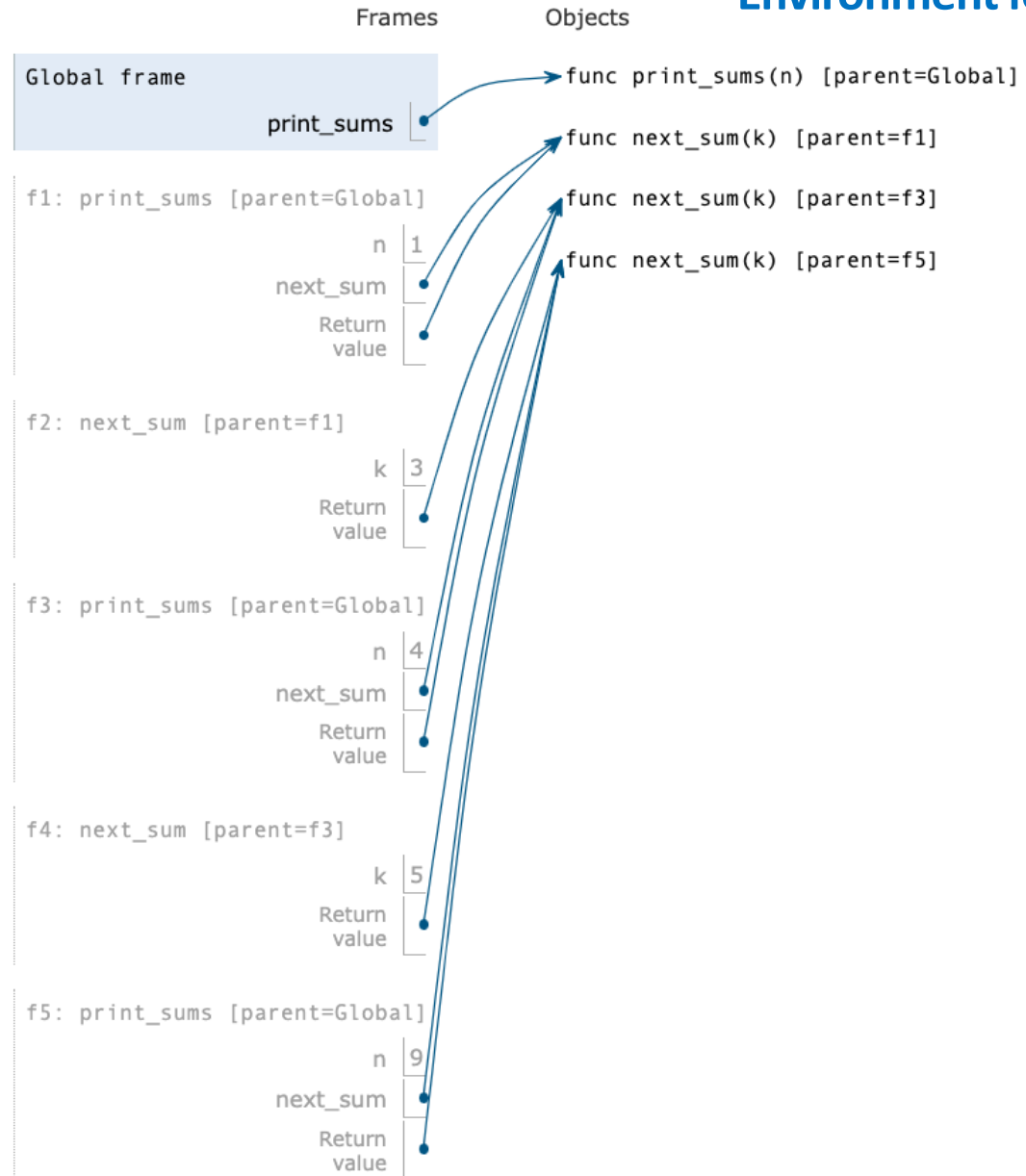
[Edit this code](#)

executed
xecute

<< First < Prev Next > Last >>

Done running (23 steps)

[ation](#)



Understanding print_sums

The call:

```
print sums (1) (3) (5)
```

- produces the same result as:

```
g1 = print sums (1)  
g2 = g1 (3)  
g2 (5)
```

Understanding print_sums

- A call to `print_sums(x)` returns a function that:
 - Prints `x` as a side-effect, and
 - Returns a function that, when called with argument `y`, will do the same thing, but with `x+y` instead of `x`.
- So, these calls will...
 - First print 1 and return `g1`,
 - which when called with 3, will print 4 (= 1+3) and return `g2`,
 - which when called with 5, will print 9 (= 4+5), and return. . . .

Passing Accumulator in the Inner Function

```
def print_sums(n): # n is accumulator
    print(n)
    def next_sum(k):
        return print_sums(n + k) # add parameter to the accumulator
    return next_sum # return inner function as next function

print_sums(1) (3) (5)
```

print_sums.py

1
4
9

Currying

SECTION 5.7

add vs. make_adder

Compare...

```
from operator import add  
add(2, 3)
```

```
def make_adder(n):  
    return lambda x: n + x  
make_adder(2)(3)
```

🤔 What's the relationship between `add(2, 3)` and `make_adder(2)(3)`?

Function currying

- **Currying:** Converting a function that takes multiple arguments into a single-argument higher-order function.
- A function that currys any two-argument function:

```
def curry2 (f) :  
    def g (x) :  
        def h (y) :  
            return f (x, y)  
        return h  
    return g
```

Function currying

- **Currying:** Converting a function that takes multiple arguments into a single-argument higher-order function.
- A function that currys any two-argument function:

```
make_adder = curry2 (add)  
make_adder (2) (3)
```

```
curry2 = lambda f: lambda x: lambda y: f (x, y)
```

```
from operator import add
def make_adder(n):
    return lambda x: n + x

msg1 = make_adder(2)(3)
print("make_adder by composite function: ", msg1)
```

```
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

```
make_adder = curry2(add)
msg2 = make_adder(2)(3)
print("make_adder by currying: ", msg2)
```

```
curry2 = lambda f: lambda x: lambda y: f(x, y)
make_adder = curry2(add)
msg3 = make_adder(2)(3)
print("make_adder by lambda currying: ", msg3)
```

curry1.py

make_adder by composite function: 5
make_adder by currying: 5
make_adder by lambda currying: 5

Why "currying"?

It's not food! ✕ 🍕 ✕ 🍕

Named after American logician Haskell Curry, but actually published first by Russian Moses Schönfinkel, based on principles by German Gottlob Frege.

See also: [Stigler's law of eponymy](#)



End of Chapter 3
