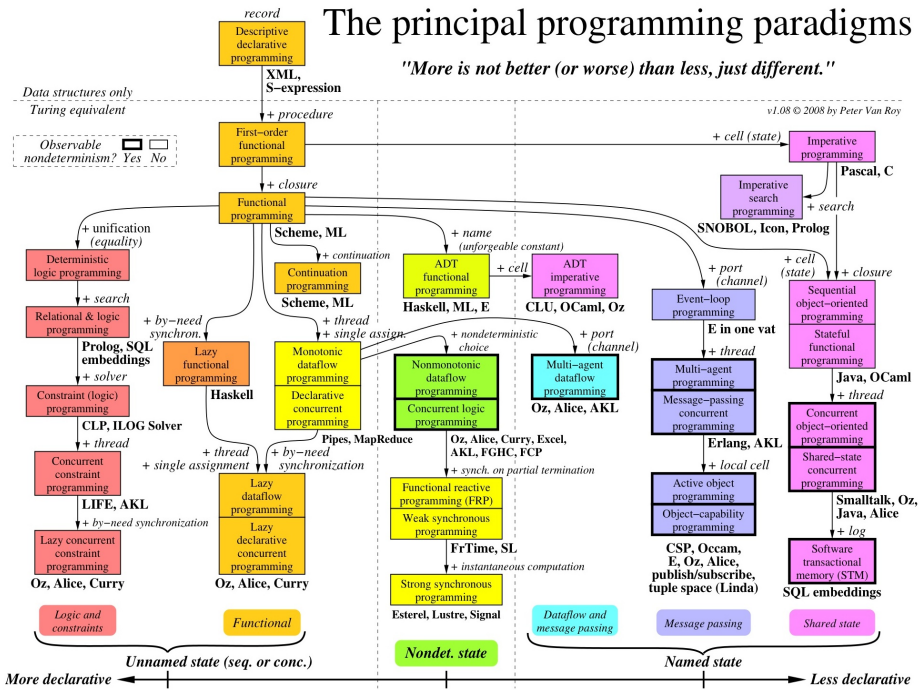
# TDDA69 Data and Program Structure
# Explicit State and Imperative Model

*Cyrille Berger*

LINKÖPING UNIVERSITY




# Lectures





# Lecture content

- Explicit state
- Environment Model
- Expression evaluation
- Function Execution
- Assignment
- Environment and Programming Language: scope
  - The Scope Trap
- Mutable Values

# Explicit state



# What is a state?

- *A state is a sequence of values in time that contains the intermediate results of a desired computation.*
- Implicit state:
  - ```
    function SumList(L, s)
    {
      define f = L.front();
      return cond(L.isEmpty(), s, SumList(T, L.tail() + s))
    }
    ```
  - Recursive calls:
    ```
    L = [1 2 3 4], s = 0
    L = [2 3 4], s = 1
    L = [3 4], s = 3
    L = [4], s = 6
    L = [], s = 10
    ```
    L and s form an *implicit* state.



# Explicit State

- An explicit state in a procedure is a state whose *lifetime extends* over more than one procedure call without being present in the procedure's *arguments*.
- Not possible with a declarative/functional model



# Cell

- A cell is an *explicit state*, it has
  - a name
  - a type
  - explicitely defined in the language
- It is stored outside the function
- Function calls are not predictable anymore
    ```

## Extension to the syntax of KL

- STATEMENT:=(...| CELL)
- CELL:='cell' IDENTIFIER (= EXPR);



# Environment Model

## Assignment and the substitution model (1/2)

```
(define (make-withdraw balance)
          (lambda (amount)
            (set! balance (-
balance amount))
                  balance))
(define W (make-withdraw 25))
(W 20)
5
(W 10)
- 5
```



## Assignment and the substitution model (2/2)

```
(define (make-withdraw balance)
          (lambda (amount)
            (set! balance (- balance
amount))
                  balance))
(define W (make-withdraw 25))
(W 20)

((make-withdraw 25) 20)
((lambda (amount) (set! balance (- 25 amount))
25) 20)
((set! balance (- 25 20)) 25)
25
```

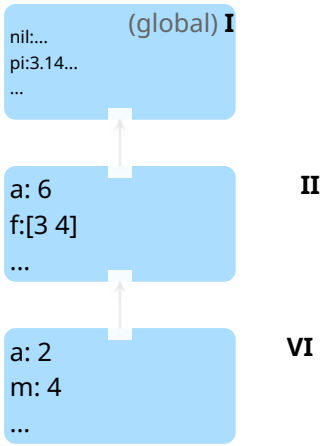
## Why doesn't the substitution model work ?

- Substitution is based on the notion that symbols are names for values
- Using set! changes symbols to places where values can be stored
- The value in such a place can change using assignment with set!
- A new model of evaluation is needed: the environment model of evaluation.



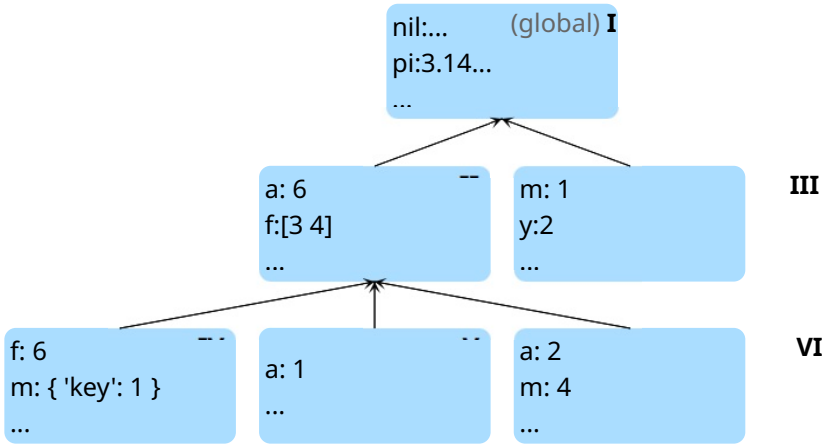
# Environment

- From the SICP Book, section 3.2:
- **Bindings** associate variables to values (in assignment expression)
- A **frame** is a set of bindings
- An **environment** is a sequence of frames
- Environments describe **contexts** where expressions are evaluated
- Every frame points to the following frame in the environment
- All environments have a **global** frame as the last frame in the sequence
- The global environment is a frame with all predefined bindings
- **Scope** the range in which a variable can be referenced






# Environment Diagram

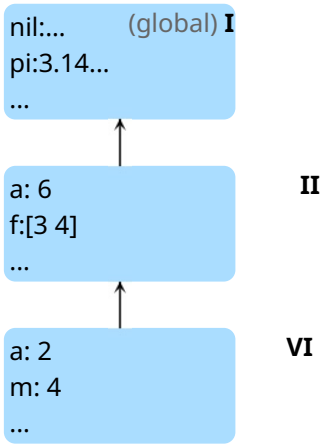





# Environments Diagrams

- Environment diagrams visualize the interpreter's process

```
def f1(a, f):
    def f2(a):
      m = a +2
     f2(2)
f1(6, [3, 4])
```



# Expression evaluation



# Evaluation of an expression

- Variables are defined in an environment
  - Variables for which there is no binding in an environment are said to be *unbound*
  - A variable can have at most *one binding per frame*
  - But a variable can have multiple *bindings in an environment*
- Expressions are associated to an environment
  - But how do we get the value of 'a'?

a plus 2



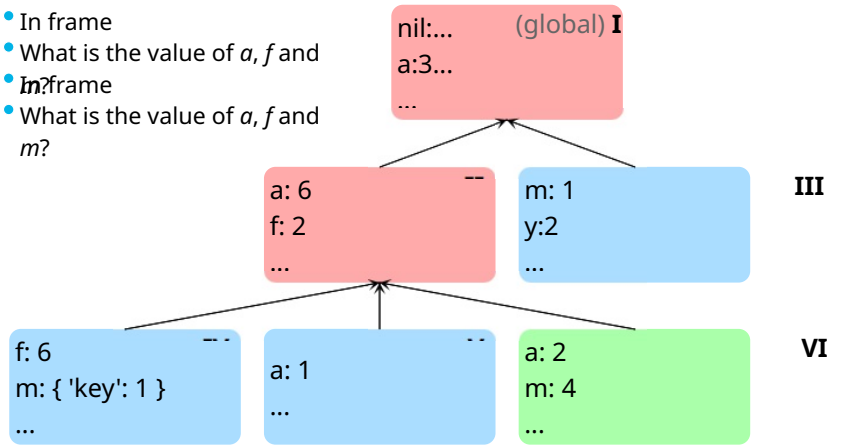
# Name look-up (1/2)

- If several bindings exist for the same variable in an environment
  - then the variable is associated to the closest binding
  - that binding is said to *shadow* the other bindings of the variable



# Name look-up (2/2)

- In frame
- What is the value of *a*, *f* and *m*?
- In frame
- What is the value of *a*, *f* and *m*?



# Function Execution



# User defined functions

- A function is stored as a lambda associated with the frame where the function was created

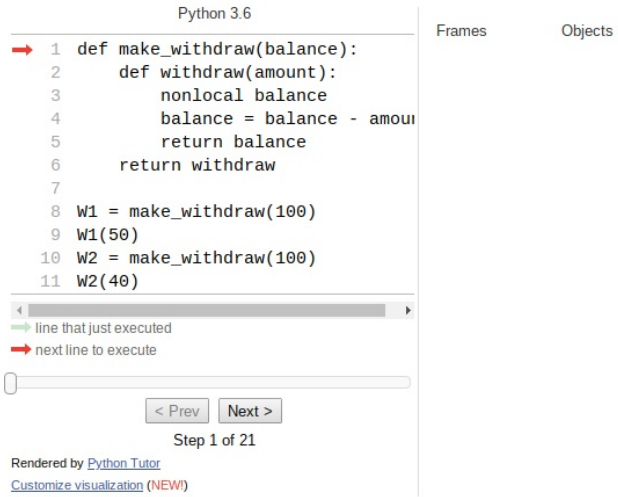


## Calling User-Defined Functions (1/2)

- Procedure for calling user-defined functions:
  - Add a local frame
  - Bind the function's formal parameters to its arguments in that frame
  - Execute the body of the function in that new frame



## Calling User-Defined Functions (2/2)



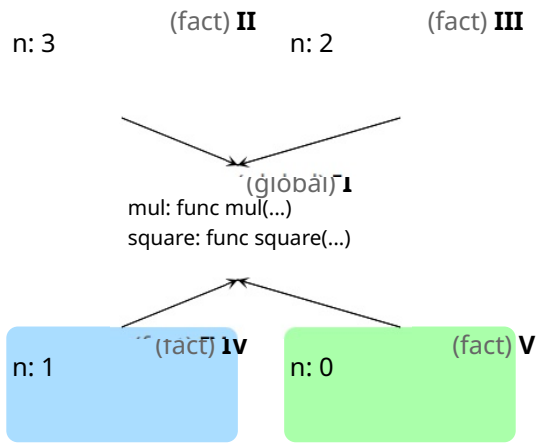
# Recursion

- The same function is called multiple time
- Different frames keep track of the different arguments in each call.
- What n evaluates to depends upon which is the current context.



# Recursion in Environment Diagram

```
def fact(n):
    if n == 0:
      return 1
    else:
      return n * fact(n-1)
fact(3)
 ↳fact(2)
  ↳fact(1)
   ↳fact(0)
```






# Assignment



# Assignment Statement

- Start in a clean state

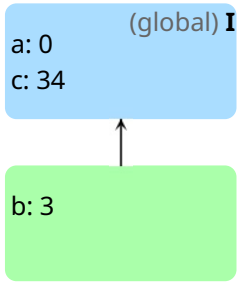
  a = 1

  b = 2

  b, a = a + b, b

# In which frame to bind?

- We are in Frame II:
  a = 2
  b = 1
- Where should the binding of a and b be done?






# In KL

```
cell a= 2;
cell b =1;
define f = function(c) {
  cell b = 3;
  a = 1; // trigger an
error
}
```



Environment and Programming Language: scope

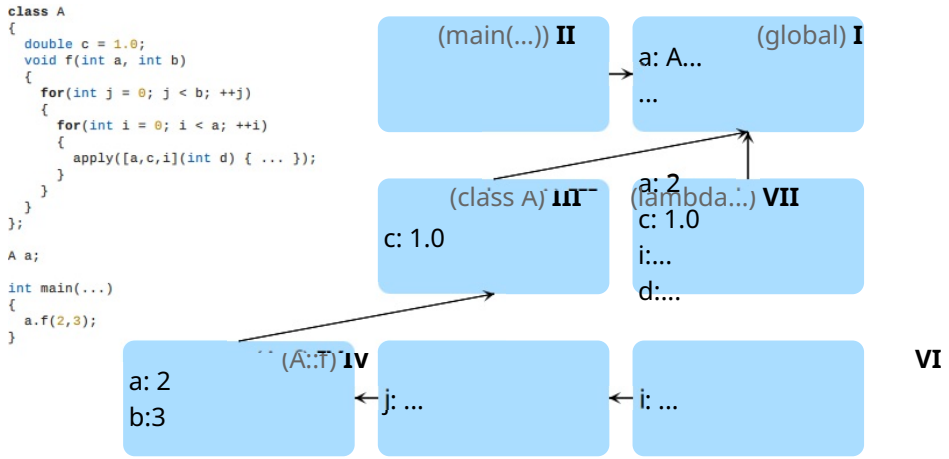


# Functions-scope vs Block-scope

- Python, JavaScript (var), Lisp... have function-scope
- C, C++, Java, JavaScript (let), KL... have block-scope
- For instance in C++:

```
int a = 1;
  {
    int a = 2
  }
  std::cout << a << std::endl;
```

# C++ and environments






# In JavaScript

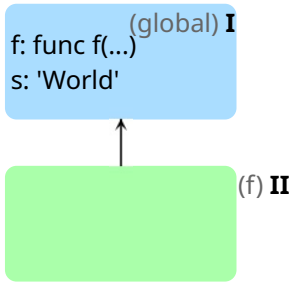
```
function f()
  {
    var s= 'Hello'
    console.log(s)
  }
  s = 'World'
  f()
  console.log(s)
```

```
function f()
  {
    console.log(s)
    s= 'Hello'
    console.log(s)
  }
  s = 'World'
  f()
  console.log(s)
```



# Local vs Global (1/4)

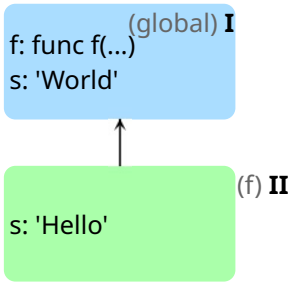
```
def f():
    print(s)
s = 'World'
f()
```






# Local vs Global (2/4)

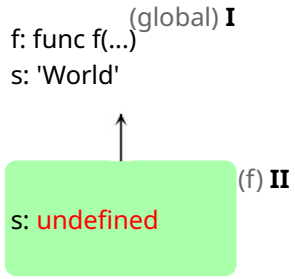
```
def f():
    s = 'Hello'
    print(s)
s = 'World'
f()
```



# Local vs Global (3/4)

```
def f():
    print(s)
    s = 'Hello'
    print(s)
s = 'World'
f()
```

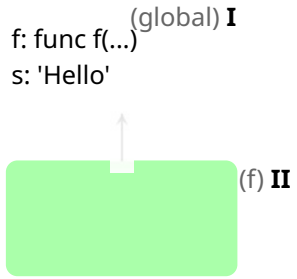
UnboundLocalError: local variable 's' referenced before assignment






# Local vs Global (4/4)

```
def f():
    global s
    print(s)
    s = 'Hello'
    print(s)
s = 'World'
f()
```






# The Scope Trap



# The Scope trap

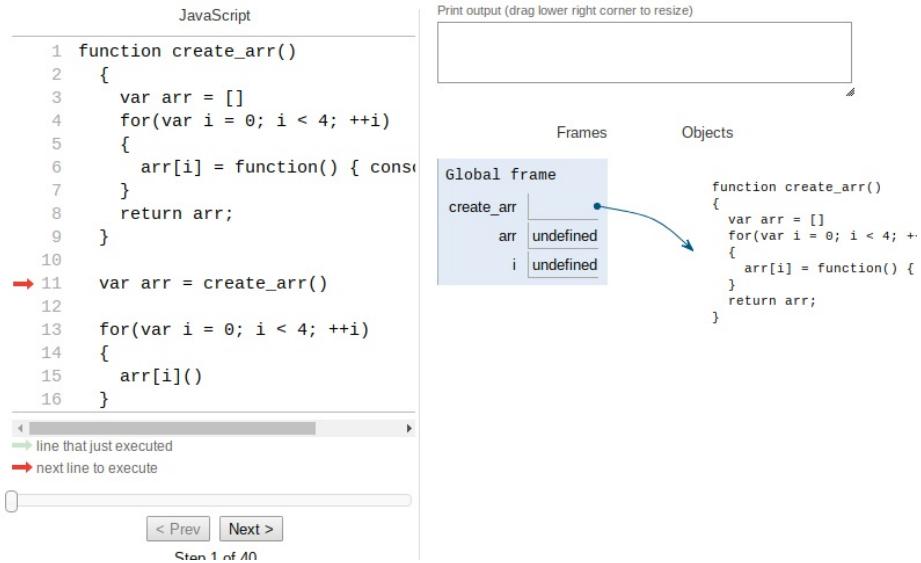
```
function create_arr()
{
  var arr = []
  for(var i = 0; i < 4; ++i)
  {
    arr[i] = function() { console.log(i) }
  }
  return arr;
}

arr = create_arr()

for(var i = 0; i < 4; ++i)
{
  arr[i]()
}
```

- What is printed?

# Calling User-Defined Functions (2/2)




# Out of the Scope Trap

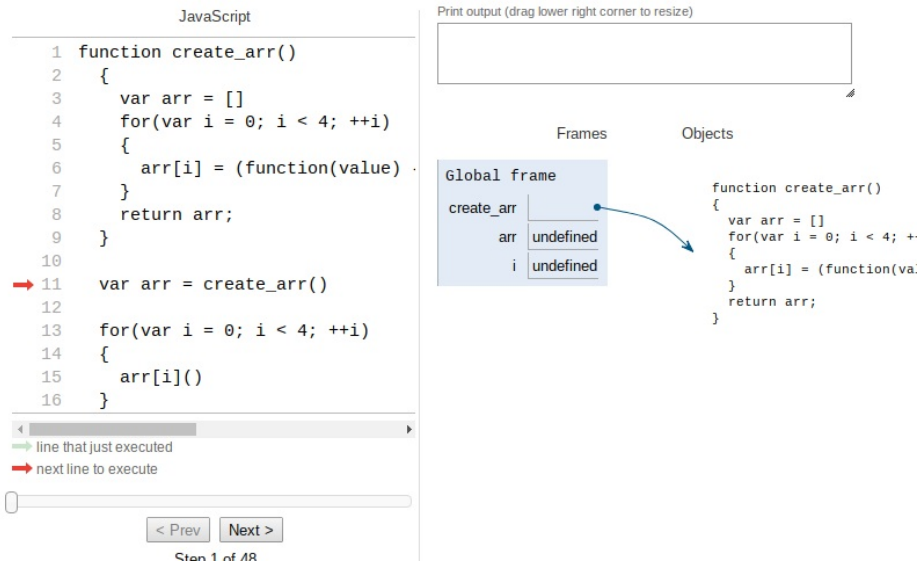
```
function create_arr()
{
  var arr = []
  for(var i = 0; i < 4; ++i)
  {
    arr[i] = (function(value) { return function() { console.log(value)
} })(i)
  }
  return arr;
}

arr = create_arr()

for(var i = 0; i < 4; ++i)
{
  arr[i]()
}
```



# Calling User-Defined Functions (2/2)



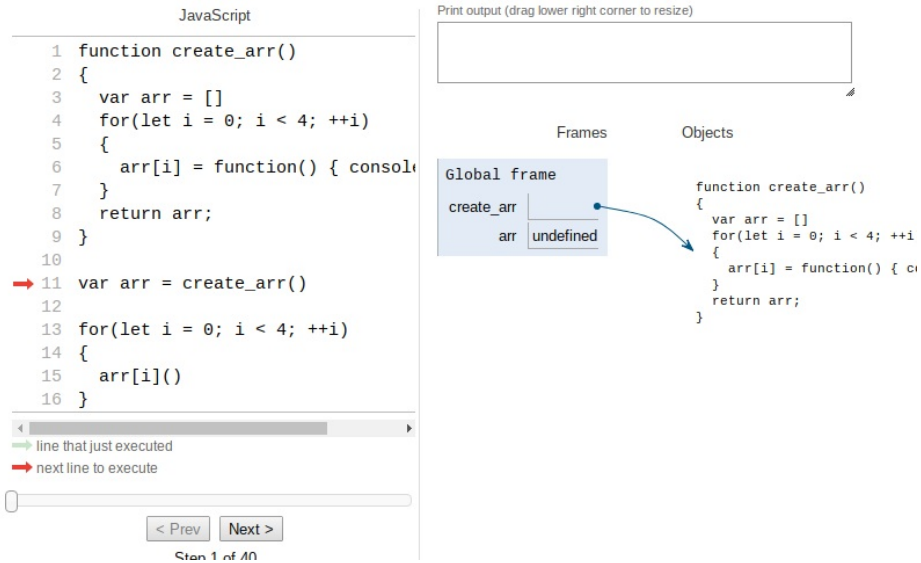

# With let... (1/2)

```
function create_arr()
{
  var arr = []
  for(let i = 0; i < 4; ++i)
  {
    arr[i] = function() { console.log(i) }
  }
  return arr;
}

arr = create_arr()

for(let i = 0; i < 4; ++i)
{
  arr[i]()
}
```

# With let (2/2)




# Mutable Values

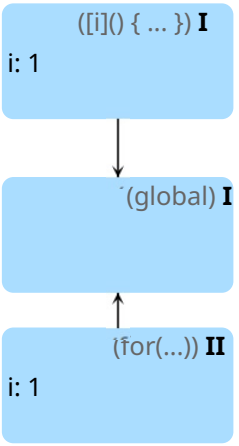


# The Scope Trap in C++

```
#include <vector>
#include <functional>
#include <iostream>

std::vector<std::function<void ()>> create_arr()
{
  std::vector<std::function<void ()>> val;
  for(int i = 0; i < 4; ++i)
  {
    val.push_back([i]() { std::cout << i << std::endl;});
  }
  return val;
}

int main(int argc, char** argv)
{
  std::vector<std::function<void ()>>  arr = create_arr();
  for(std::function<void ()> v : arr)
  {
    v();
  }
}
```




- In C++, captured variables are copied in the lambda frame

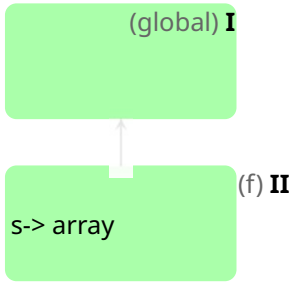


# States

- An assignment changes the value of a variable
- All names are affected by the mutation
- ```
  a = Object()
  b = a
  a.v = 1
  print(b.v)
  ```
  ```

## Mutation within a function call (1/2)

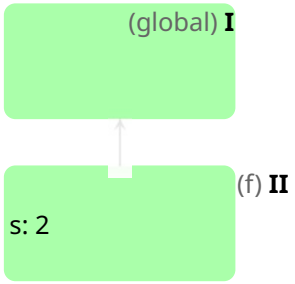
```
array = [1, 2, 3,
4]
len(array) -> 4
f(array)
len(array) -> 2
def f(s):
    s.pop()
    s.pop()
```






## Mutation within a function call, always? (1/2)

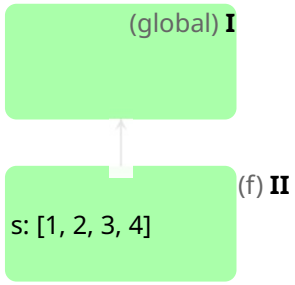
```
value = 2
print(value) -> 2
f(value)
print(value) -> 2
def f(s):
    s = 1
```






## Mutation within a function call, always? (2/2)

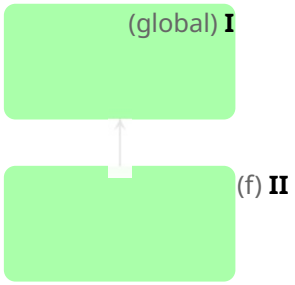
```
array = [1, 2, 3,
4]
len(array) -> 4
f(array)
len(array) -> 4
def f(s):
    s = list(s)
    s.pop()
    s.pop()
```






## Mutation within a function call (2/2)

```
array = [1, 2, 3,
4]
len(array) -> 4
f()
len(array) -> 2
def f():
    global array
    array.pop()
    array.pop()
```



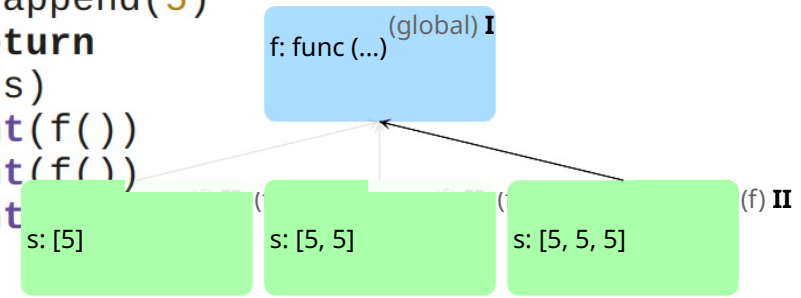
# Drawbacks of mutability

- `pi = 2`
- What happen if you do this?
  - ```
    def f(s=[]):
        s.append(5)
        return len(s)
    print(f())
    print(f())
    print(f())
    ```



# Mutable Default arguments

- ```
  def
  f(s=[]):
      s.append(5)
      return
  len(s)
  print(f())
  print(f())
  print
  ```






# Comparison (1/2)

- In Python:
- Value comparison

```
a = [1, 2]
b = [1, 2]
c = a
print(a == b) -> true
print(a == c) -> true
```

- Identity comparison

```
print(a is b) -> false
print(a is c) -> true
```



# Comparison (2/2)

- In JavaScript:
- Value comparison

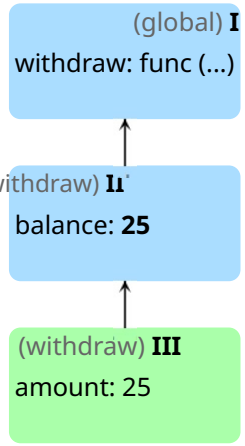
```
a = 5
b = "5"
```

- Identity comparison

```
console.log(a == b) -> true
console.log(a === b) -> false
```

# Mutable Functions

- A function whose behavior varies over time
- Example:
  - ```
    def make_withdraw(balance):
        def withdraw(amount):
            balance = balance - amount
            return balance
        return withdraw
    withdraw = make_withdraw(100)
    withdraw(25) -> 75
    withdraw(25) -> 50
    withdraw(25) -> 25
    ```






# Conclusion

- Cell
- Environment model
- Problems with scopes and global
- Mutation
    ```