Functions in Python
Part 2

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Concept of symbol table
- Scope of identifiers
  - local vs. global scope
  - scope of formal parameters
  - local functions

Identifier that are local to the function

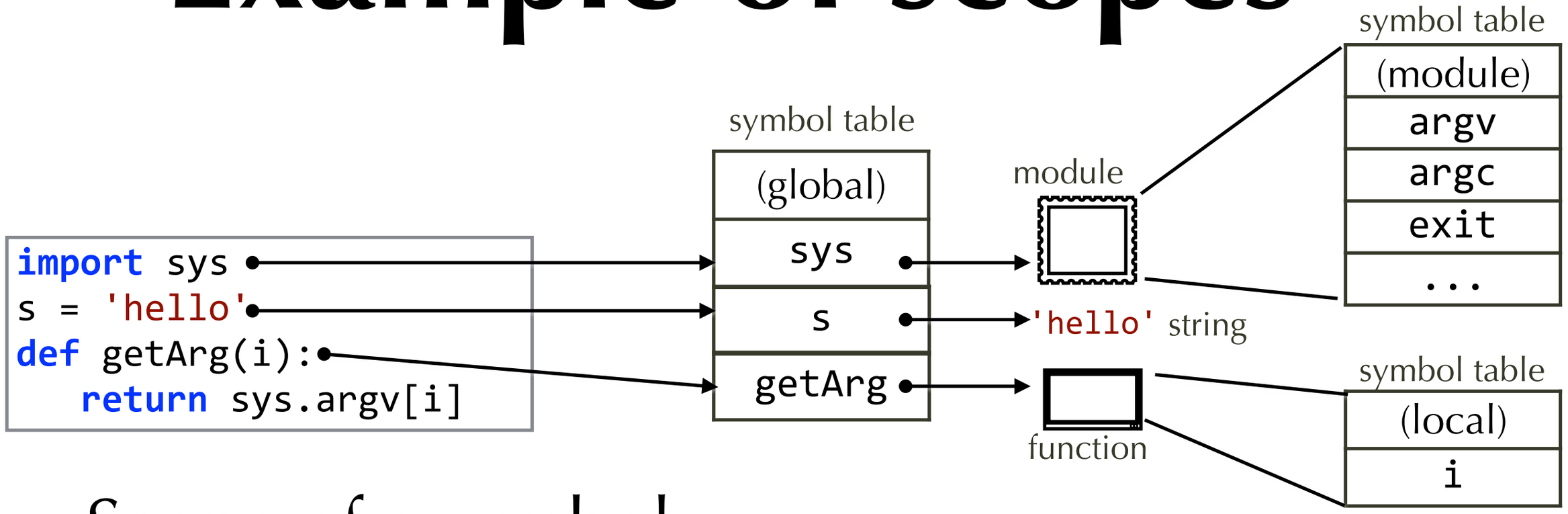
- Formal parameters
  - `rate`, `priceDict` are local to this function
- Local variables
  - `sum`, `i` are local to this function
- The same name may be defined outside this function, but they are entirely unrelated
- Local names are deleted on exit from function

```
def totalTax(rate, **priceDict):
    sum = 0
    for i in priceDict.values():
        sum = sum + i
    return(1 + rate) * sum
```

Symbol table

- a data structure for tracking identifiers and what they refer to
  - i.e., a `dict` data structure
- Python runtime system maintains several symbol tables
  - top-level ("global"), each module, each function call, each data object...

Example of scopes



- Scope of a symbol
  - its "visible region"
  - its "lifetime" (from created to destroyed)

Examine the symbol table using `dir()`

- `dir(`*obj*`)` returns a list of identifiers defined in the obj's symbol table
- `dir()` returns a list of identifiers defined in the current scope

```
>>> A = 20
>>> import sys
>>> dir()
['A', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'sys']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__',
'__excepthook__', '__interactivehook__', '__loader__', '__name__',
'__package__', '__spec__', '__stderr__', '__stdin__', ...]
```

Deletion

- deleting a name
  - the identifiers becomes no longer defined by assignment or by function definition
- object reclamation
  - if data object is no longer accessible, its space may be "recycled"

Deletion

- a name exists until
  - the block exits, or the name is deleted by `del`
- a data object may also be deleted

```
>>> A = [1, 2, 3]   # A refers to the list [1, 2, 3]
>>> B = A           # B refers to the same list as A does
>>> del A[1]        # delete element from list
>>> A               # middle element of list is deleted
[1, 3]
>>> del A           # deletes the name A, not the object
>>> B
[1, 3]              # the modified list that A used to refer to
>>> A
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined
```

Names are local to functions

- created on first assignment in the function
- automatically deleted on exiting function
  - However, data can be retained!
- Names outside the function are unaffected

```
>>> x = 3               # x is a top-level name
>>> def f():            # f also a top-level name
...     x = 4           # This x is local to function f!
...     print(x)        # deleted upon exiting function f
...
>>> f()                 # call function f, which makes local x and deletes
4
>>> x                   # the top-level x, unrelated to f's x
3
```

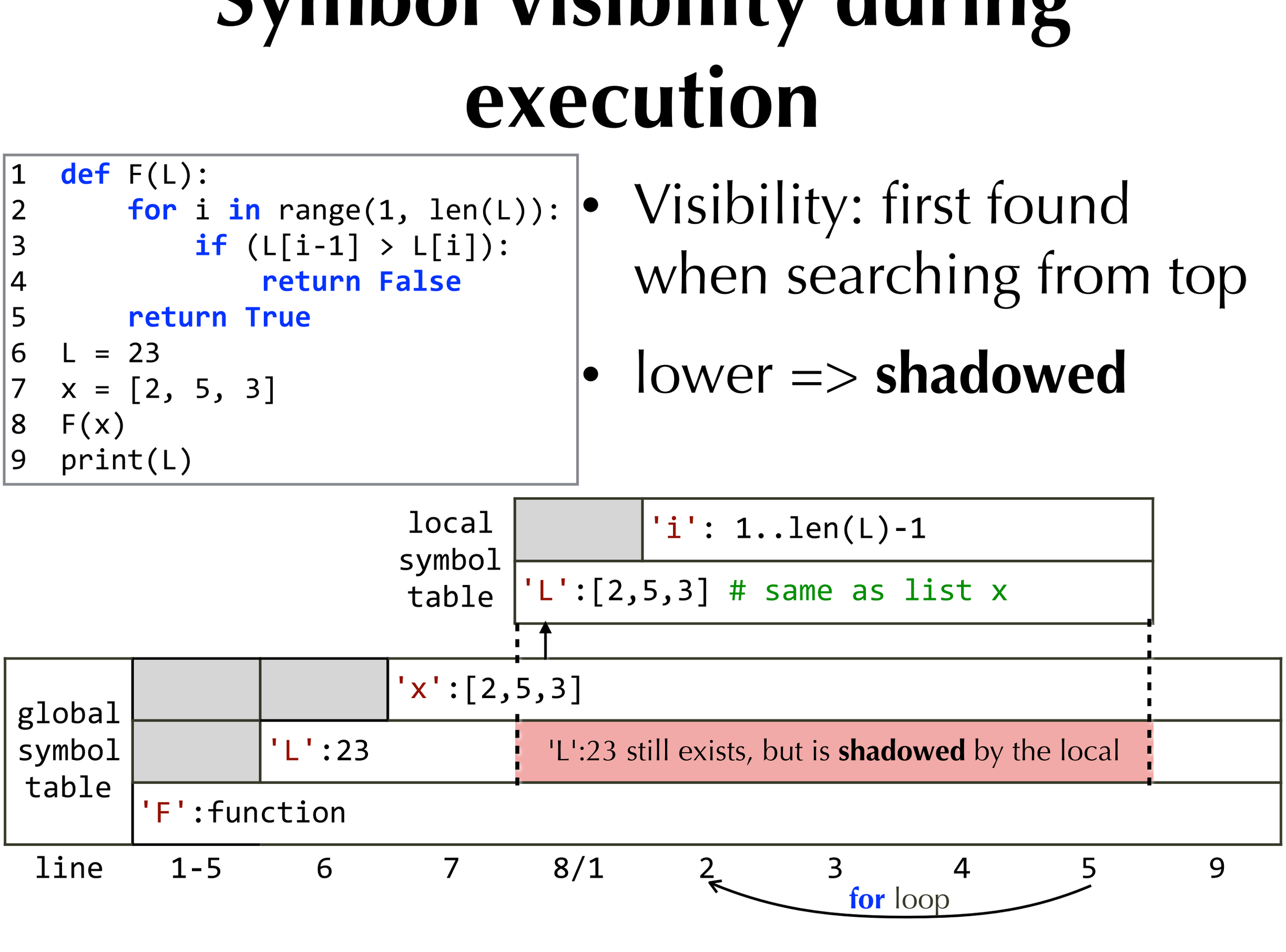
Implementation of Symbol Tables

- Dictionary
  - identifier = key, association = value: add, modify, delete
- Global symbol table:
  - used throughout program execution
- Local symbol table
  - Last-in, first-out (LIFO) order of symbol table
- Symbol lookup
  - Local: try local ST first; if not found, try global.
  - A local may **shadow** a global symbol

Symbol visibility during execution

```
1  def F(L):
2      for i in range(1, len(L)):
3          if (L[i-1] > L[i]):
4              return False
5      return True
6  L = 23
7  x = [2, 5, 3]
8  F(x)
9  print(L)
```

- Visibility: first found when searching from top
- lower => **shadowed**



Rules for identifier binding

- Look-up rule: two levels
  - look in the symbol table in its local scope, if any
  - if not found, look in the global symbol table.
- Local-definition rule
  - Always **create** a new definition in the "most local" symbol table
- Consistency rule:
  - lookup binding and definition binding must be consistent, or else it is an error.

Rule 1: two-level lookup

- Look in local if available or found
- if not, look in global.

```
1  a = 3
2  def F():
3      print(a)
4  F()
```

- line 1 adds `'a':3` to the global symbol table
- line 2-3 define `'F'` as function in global symbol table
- line 4 calls F, goes to line 3, `'a'` not in local symbol table
  => look in global symbol table, found `'a':3`
  => prints 3

Rule 2: new definition goes to the most local scope

code

```
1  a = 3
2  def F():
3      a = 5
4      print(a)
5  F()
6  print(a)
```

output

```
5
3
```

- line 1 adds `'a':3` to global symbol table
- line 2-4 def `'F'` in global
- line 5 calls F, goes to line 3,
  `'a'` not in local symbol table => adds entry
  `'a':5` in local! (shadows global `'a'`)
- line 4 prints `5` (local value)
- upon return, destroy local symbol table
  (global `'a'` becomes visible again)
- line 6 looks up `'a'`, found global `'a':3`
  => prints `3` (global value)

Rule 3: consistent binding

code

```
1  a = 3
2  def F():
3      print(a)
4      a = 5
5  F()
6  print(a)
```

- What if we swap lines 3 and 4 from previous example?
  - line 3 lookup => global `'a'`:3
  - line 4 define => local `'a'`:5
    => violation of consistent binding rule!

You will get an error at runtime:

```
UnboundLocalError: local variable 'a'
referenced before assignment
```

`global` keyword

- syntax:
  `global` *var1*`,` *var2*`,` ...
- Forces consistent binding to be global when you are in a local scope

code

```
1  a = 3
2  def F():
3      global a
4      print(a)
5      a = 5
6  F()
7  print(a)
```

bind to the global one, don't create it locally

output

```
3
5
```

Example use of Global: totalWithTax

- Two ways to specify amount to total in list:
  - by numeric amount
  - by item name (string) - to lookup amount in dict
- Example, want a function like this

```
>>> totalWithTax(rate=0.05, apple=20, orange=15, guava=12)
49.35                    # remembers these parameters for next time
>>> totalWithTax('apple', 'guava')
33.6
```

Source code for totalWithTax version 1

```
#!/usr/bin/env python3
D = { 'rate': 0.0 }

def totalWithTax(*names, **kv):
    global D
    total = 0.0
    for name in names:
        total += D[name]

    for kw, val in kv.items():
        D[kw] = val  # overwrite dict entry
        if kw != 'rate':
            total += val

    return total * (1 + D['rate'])
```

Feature 1: What if we also want to include numeric arguments?

- In current implementation

```
>>> totalWithTax(rate=0.05, apple=20, orange=15, guava=12)
49.35                    # remembers these parameters for next time
>>> totalWithTax('apple', 'guava')
33.6
>>> totalWithTax(23, 45, 'oranges', mango=60)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in totalWithTax
KeyError: 23
```

- Reason:
  - we attempted to look up D[23], even though we should just use the value 23

Feature 2: automated testing

- Why? too much trouble to keep testing code by typing
  - code can include test case, report error if wrong, but silent if no error => use **`assert`**!

- Test code only if run the file as top-level;
  - don't test if just imported

```
if __name__ == '__main__':
    assert totalWithTax(rate=0.05, apple=20, oranges=15, guava=12)) \
           == 49.35
    assert totalWithTax('apple', 'guava') == 33.6
    assert totalWithTax(23, 45, 'oranges', mango=60) == 150.15
```

Source Code version 2: accept numerical arguments

```
#!/usr/bin/env python3
D = { 'rate': 0.0 }
def totalWithTax(*names, **kv):
    global D
    total = 0.0
    for name in names:
        if type(name) in {int, float}:
            total += name
        else:
            total += D[name]
    for kw, val in kv.items():
        D[kw] = val  # overwrite dict entry
        if kw != 'rate':
            total += val
    return total * (1 + D['rate'])
if __name__ == '__main__':
    assert totalWithTax(rate=0.05, apple=20, oranges=15, guava=12)) \
           == 49.35
    assert totalWithTax('apple', 'guava') == 33.6
    assert totalWithTax(23, 45, 'oranges', mango=60) == 150.15
```