



Python for Astronomers

Functions and modules

Exercise

- The files HD152200.uvspec and HD152234.uvspec in /home/mmetz contain FUSE spectra (thanks to Ole !) of the star HD* as ASCII data (wavelength, count rate).
 - i) Read the data, normalise the count rates, and plot both spectra into one subplot.
 - ii) Create a subplot below the first one and plot binned spectra (use numpy to bin the data)

Defining functions

```
>>> def fib(n):  
...     """Print Fibonacci numbers up to n"""  
...     a, b = 0, 1  
...     print a,  
...     while b <= n:  
...         print b,  
...         a, b = b, a + b  
  
>>> fib(2000)    # Execute function for integer  
>>> fib(1000.)  # for float  
>>> fib("abc")  # and for string
```

return statement

```
>>> def fib2(n):
...     fibs, newfib = [0, 1], 1
...     while newfib <= n:
...         fibs.append(newfib)
...         newfib = fibs[-1] + fibs[-2]
...     return fibs

>>> f2000 = fib2(2000)
>>> print f2000

>>> f1000 = fib(1000)    # Function fib has no return!
>>> print f1000         # None returned
```

Recursive functions

```
>>> def fact(n):  
...     if n > 0:  
...         return n * fact(n-1) # Recursive call  
...     return 1 # exits function returning 1  
  
>>> print fact(100)  
>>> print fact(1000)
```

Excercise: Determine the maximum recursion depth

Solution

```
>>> n = 0
>>> while fact(n):
...     print n
...     n += 1
```

Calling functions

```
>>> def f(x, y, z):
```

```
...     print x, y, z
```

```
>>> # Using positional arguments:
```

```
>>> f("Ham", 17+4, [1,2,3])
```

```
>>> f(17+4, [1,2,3], "Ham")
```

```
>>> # Using keyword arguments:
```

```
>>> f(x="Spam", y=13, z=range(3))
```

```
>>> f(y=13, z=range(3), x="Spam")
```

```
>>> # Using positional and keyword arguments:
```

```
>>> f(1, z="last", y=[2])
```

Default arguments

```
>>> def f(x=1, y=2, z=3):
```

```
...     print x, y, z
```

```
>>> f()
```

```
>>> f(42, 13)
```

```
>>> f(z="Three")
```

```
>>> def f(x, y, z=None):
```

```
...     if z is None:
```

```
...         doSomething(x, y)
```

```
...     else:
```

```
...         doSomethingElse(x, y, z)
```

Excercises: Function calls

```
>>> def f(x, y, z):  
...     print x, y, z  
  
>>> # Test the following function calls:  
>>> f(1, 2)  
>>> f(1, 2, 3, 4)  
>>> f(x=1, y=2, bla=3)  
>>> f(1, x=2, y=3, z=4)  
>>> f(x=1, 2, 3)  
  
>>> # Repeat with  
>>> def f(x="a", y="b", z="c"):  
...     print x, y, z
```

Local variables

```
>>> import math

>>> def area(r):
...     """Area of circle with radius r"""
...     return math.pi * r**2    # Name math is known!

>>> def volume(r, h):
...     """Vol. of cylinder with radius r, height h"""
...     return area(r) * h      # Name area is known!

>>> volume(1., 2.)
```

Local variables

```
>>> def f1():
...     print x          # Use variable x

>>> def f2():
...     x = "local x"   # Assign variable x
...     print x

>>> x = "global x"     # Same name x as in functions

>>> f1()                # "global x"

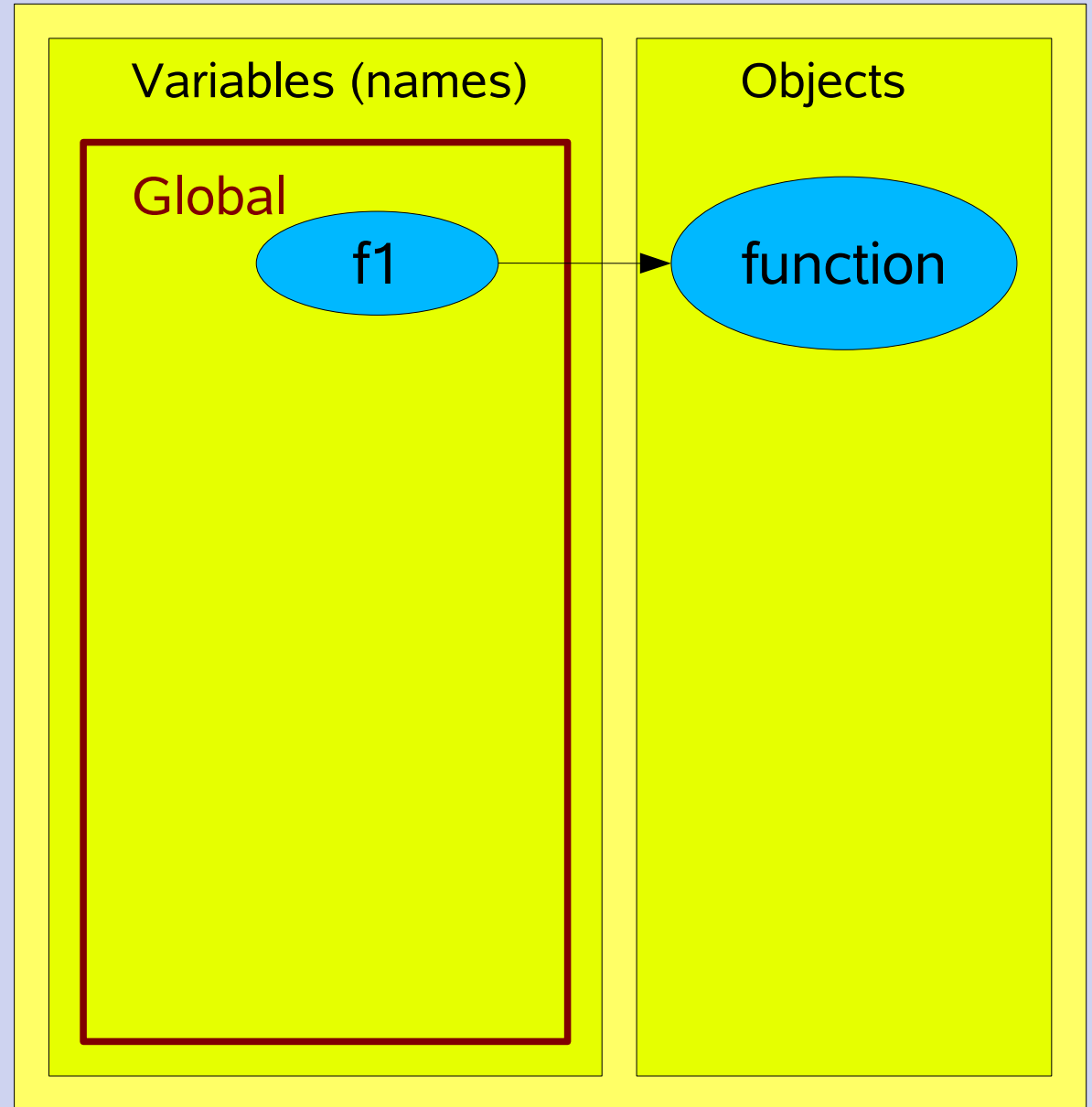
>>> print x             # "global x"

>>> f2()                # "local x"

>>> print x             # still "global x"
```

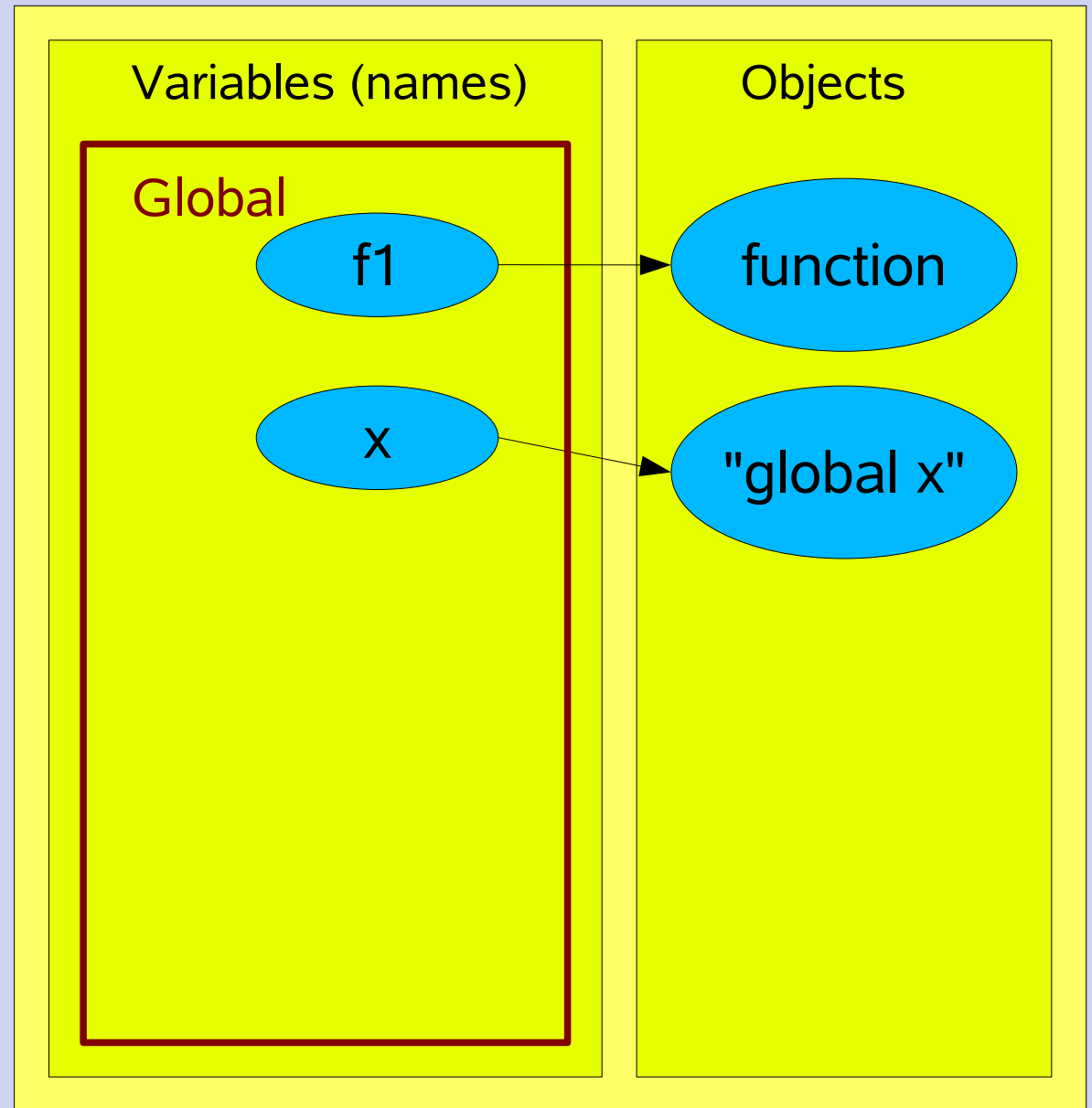
Function namespace

```
>>> def f1():  
...     print x
```



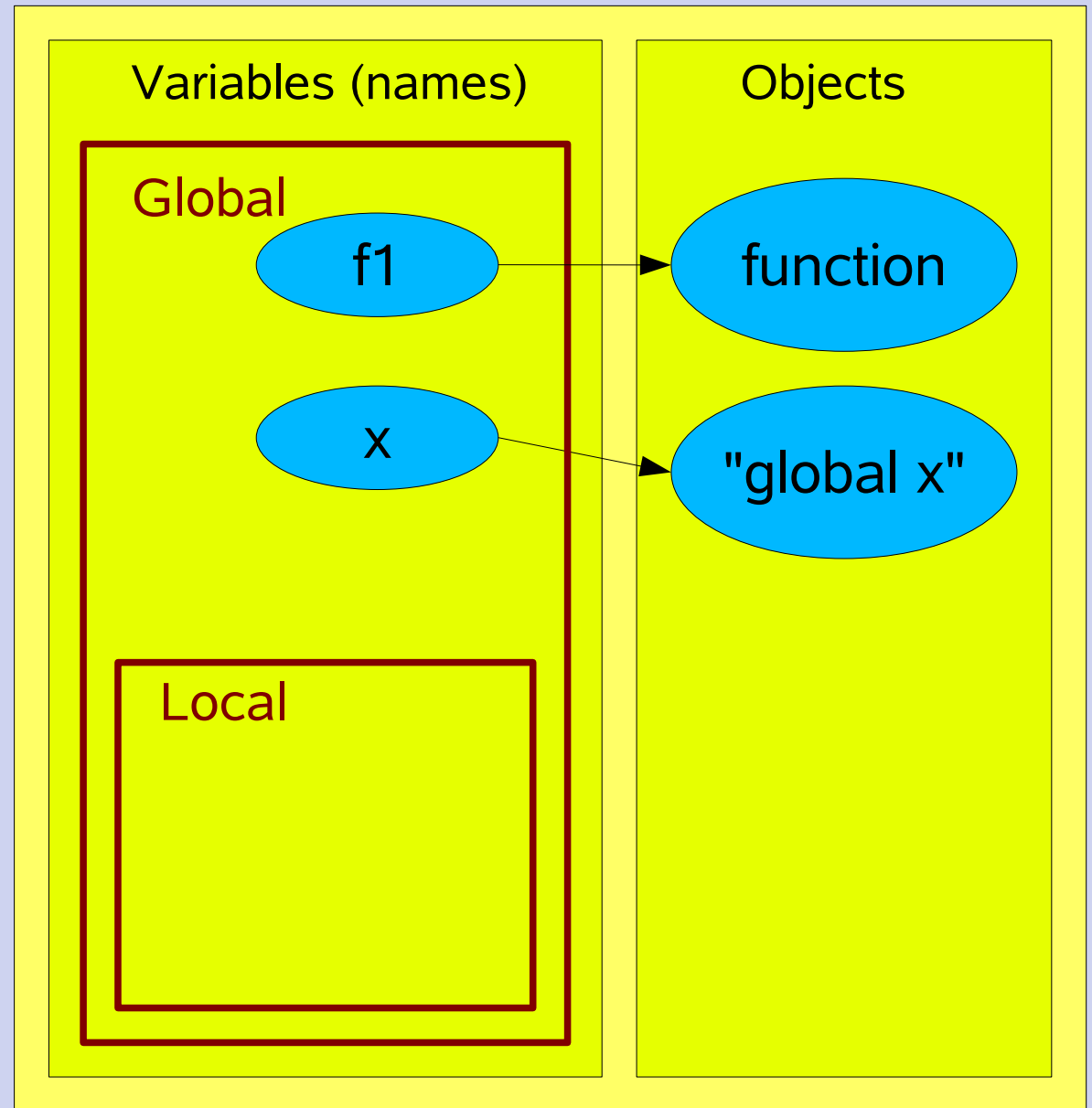
Function namespace

```
>>> def f1():  
...     print x  
  
>>> x = "global x"
```



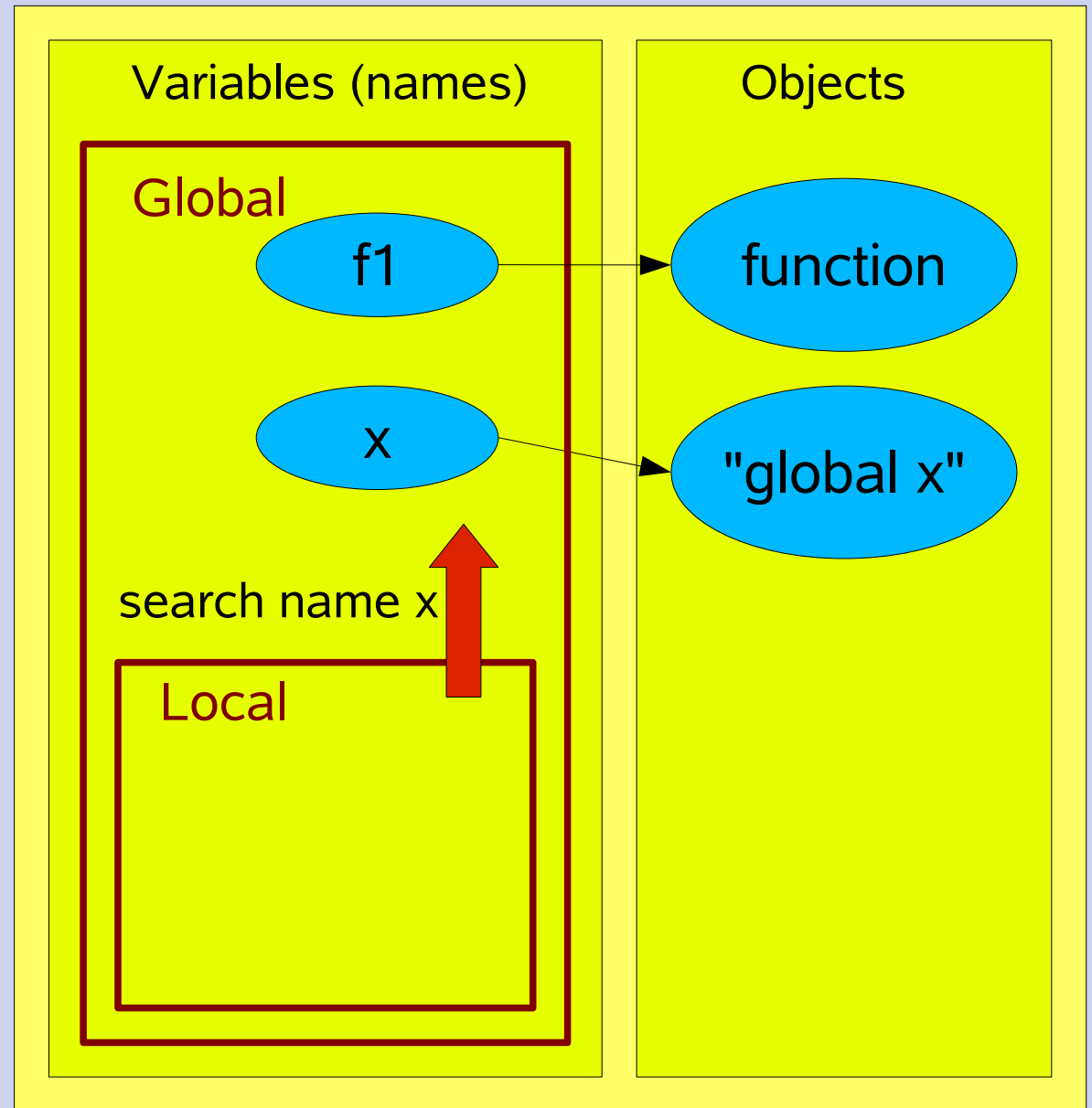
Function namespace

```
>>> def f1():  
...     print x  
  
>>> x = "global x"  
  
>>> f1()
```



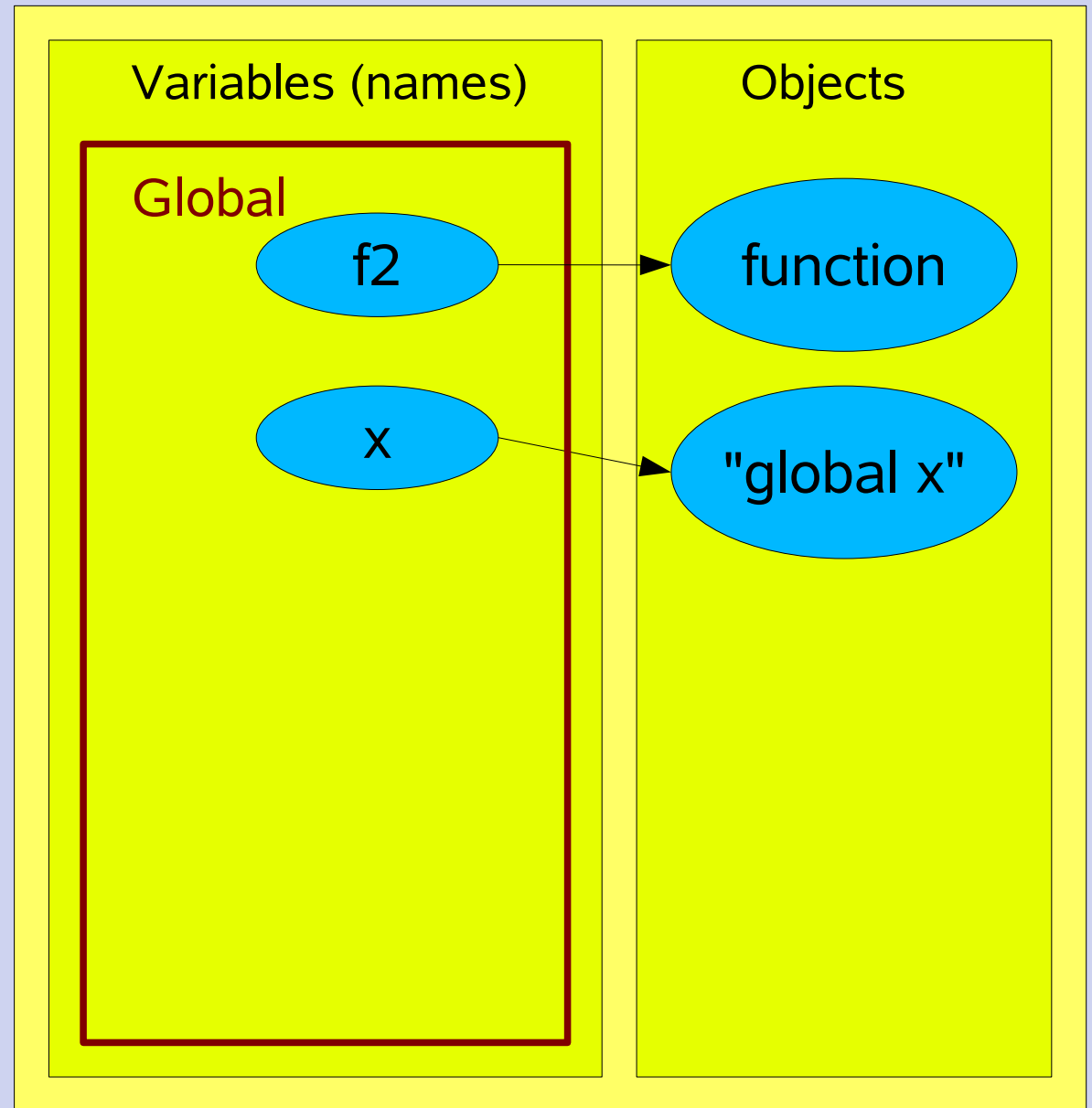
Function namespace

```
>>> def f1():  
...     print x  
  
>>> x = "global x"  
  
>>> f1()
```



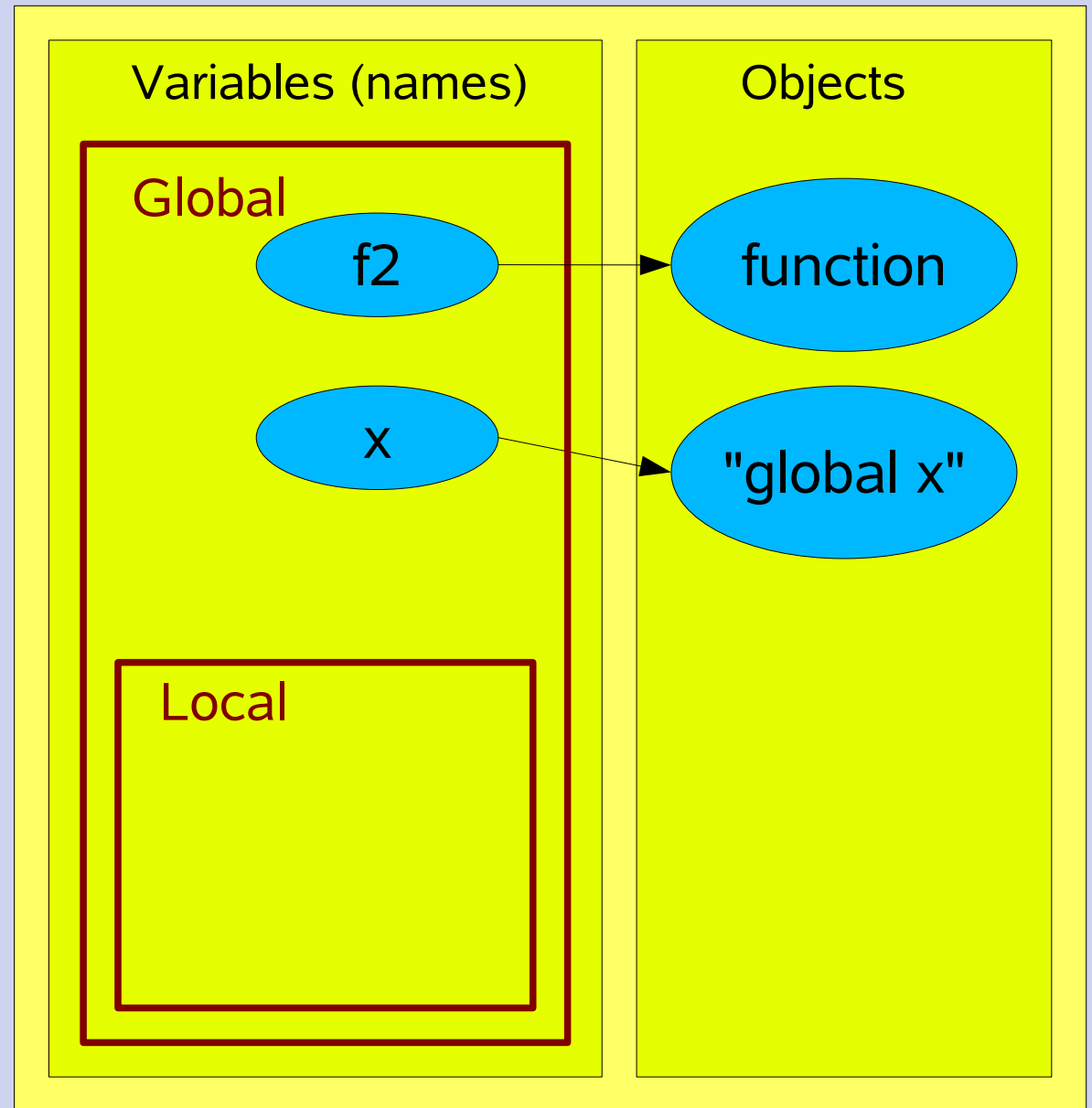
Function namespace

```
>>> def f2():  
...     x = "local x"  
...     print x  
  
>>> x = "global x"
```



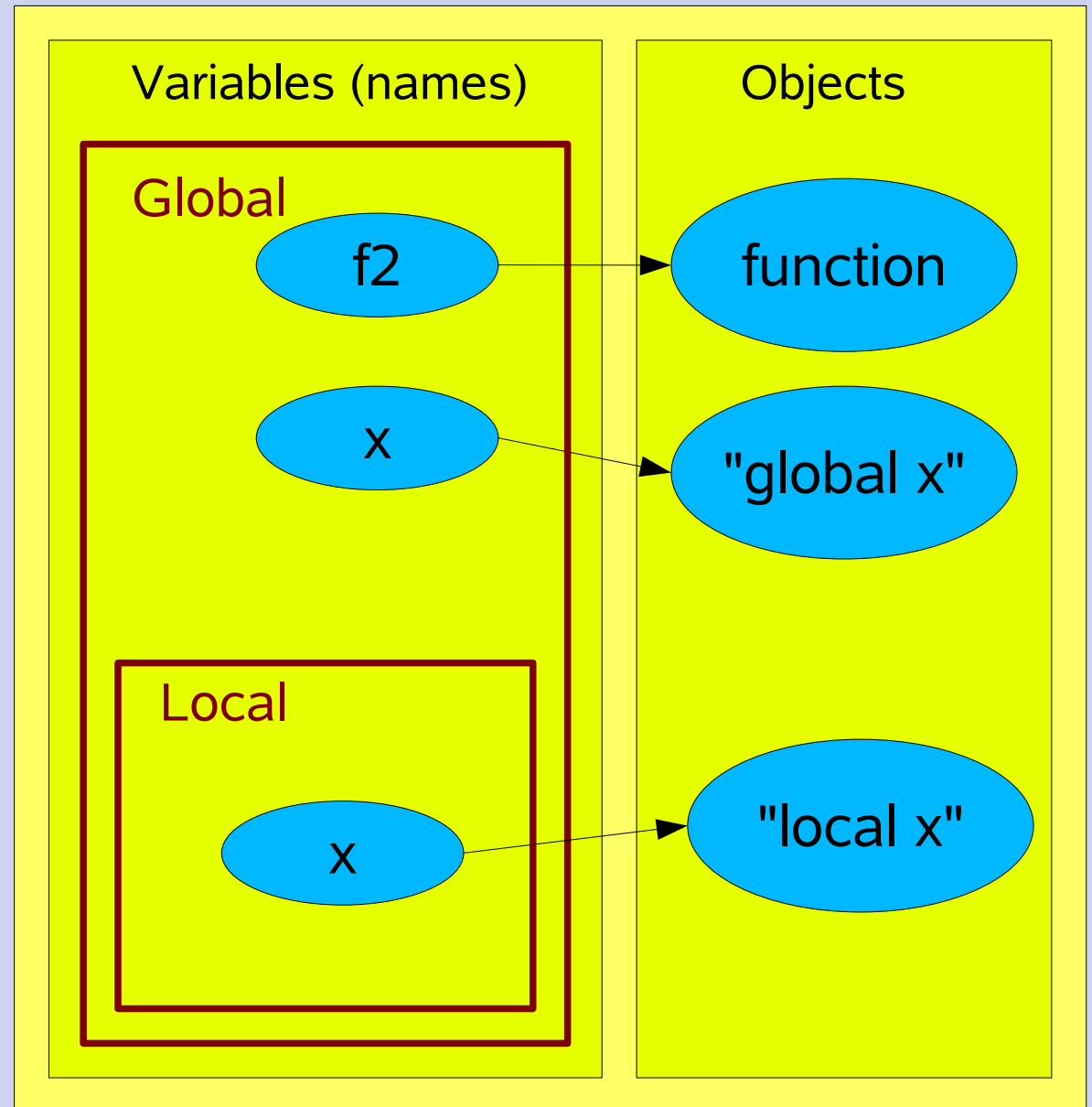
Function namespace

```
>>> def f2():  
...     x = "local x"  
...     print x  
  
>>> x = "global x"  
  
>>> f2()
```



Function namespace

```
>>> def f2():  
...     x = "local x"  
...     print x  
  
>>> x = "global x"  
  
>>> f2()
```



Scope rules

Functions provide a nested namespace (scope)

- Name ***references*** search four scopes:
 - ***L***: the function's local scope
 - ***E***: the scope of enclosing functions
 - ***G***: the (module's) global scope
 - ***B***: the built-in scope (e.g. range function)
- Name ***assignments*** create local names

The global statement

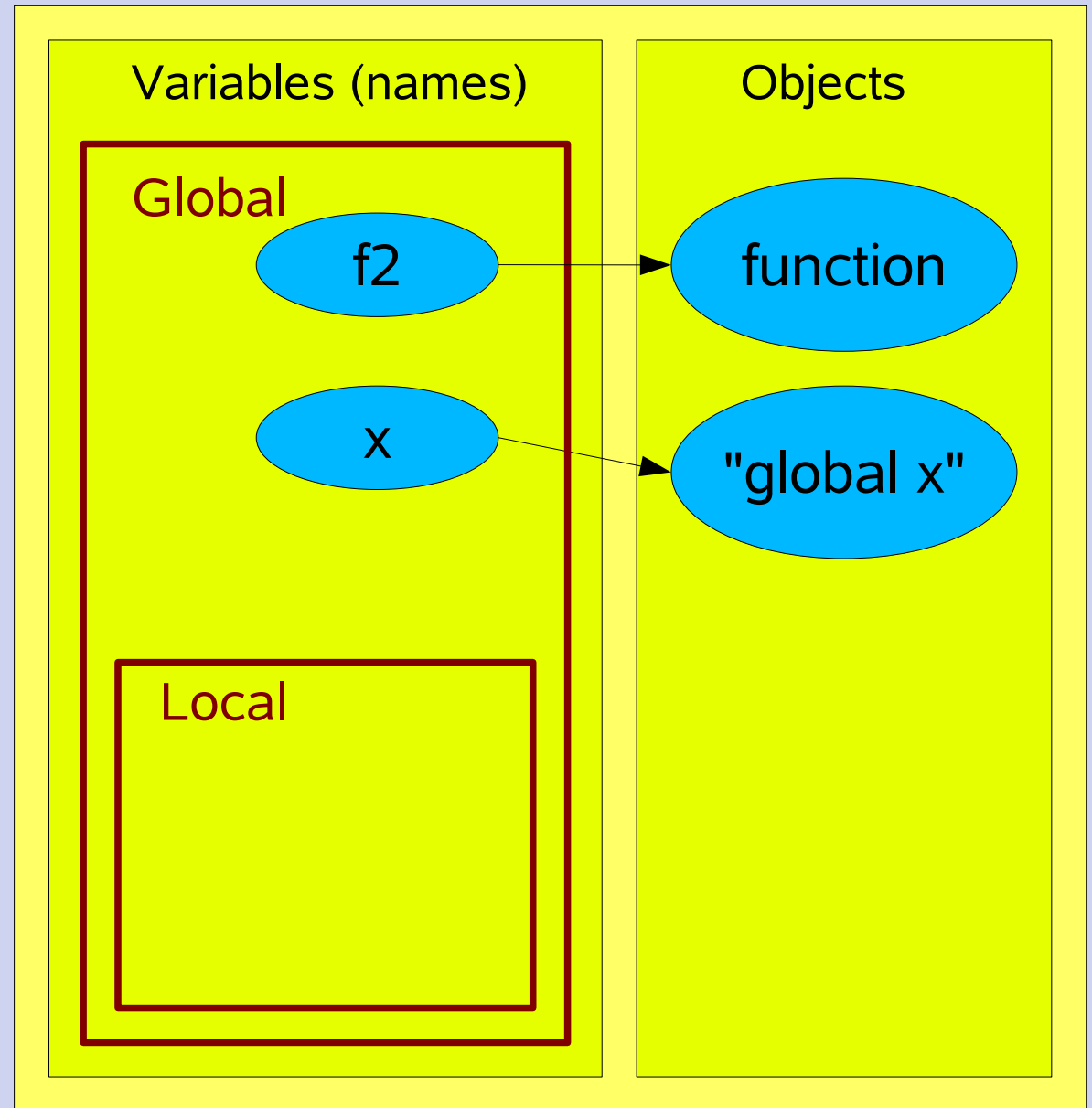
```
>>> def f3():
...     global x
...     x = "local x"
...     print x

>>> x = "global x"

>>> print x           # "global x"
>>> f3()             # "local x"
>>> print x           # now "local x"
```

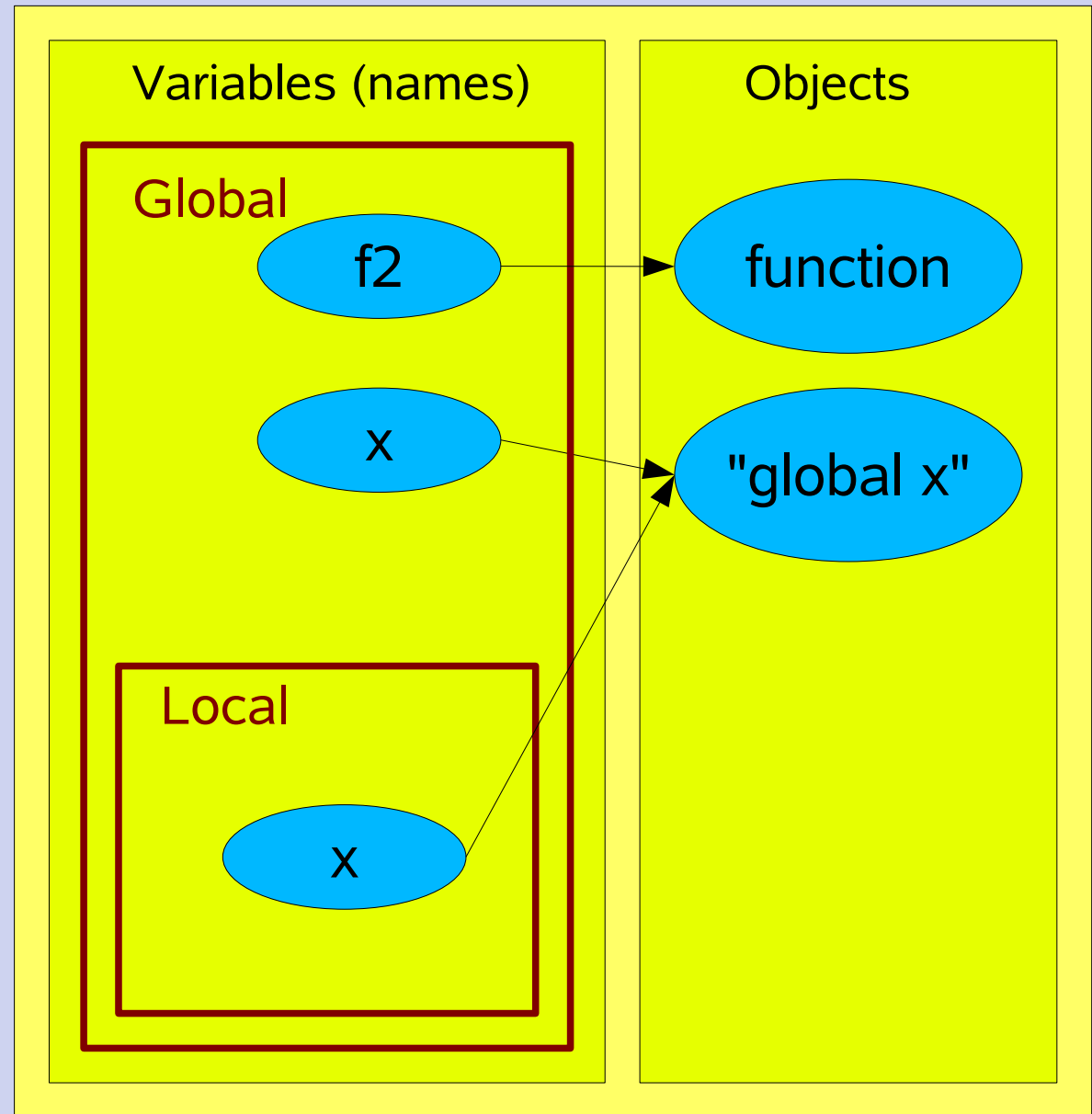
Function namespace

```
>>> def f3():  
...     global x  
...     x = "local x"  
...     print x  
  
>>> x = "global x"  
  
>>> f3()
```



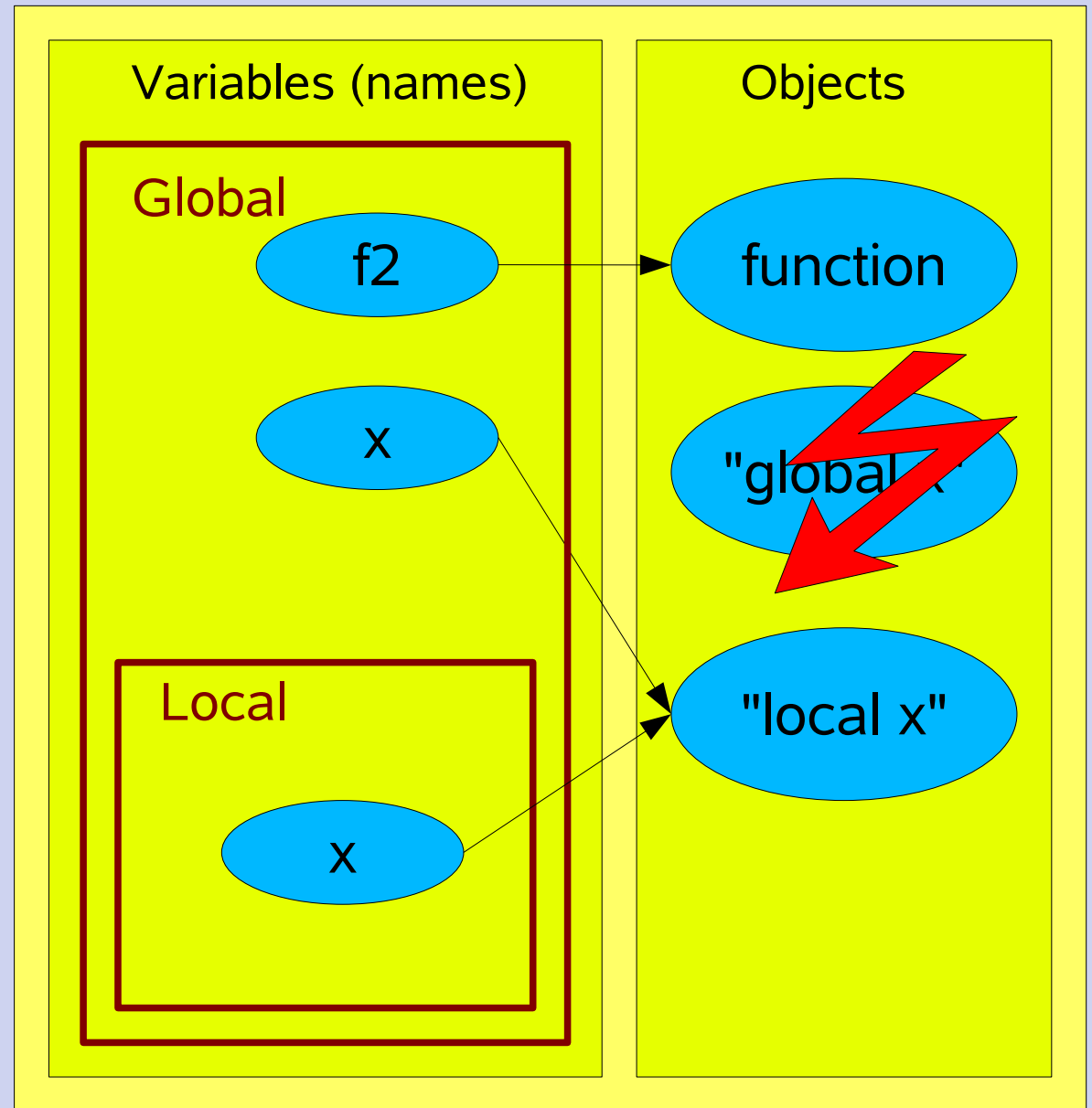
Function namespace

```
>>> def f3():  
...     global x  
...     x = "local x"  
...     print x  
  
>>> x = "global x"  
  
  
>>> f3()
```



Function namespace

```
>>> def f3():  
...     global x  
...     x = "local x"  
...     print x  
  
>>> x = "global x"  
  
  
  
>>> f3()
```



Argument passing - Experiments

```
>>> def twice1(a):  
...     a = 2 * a  
  
>>> x = 42  
>>> twice1(x)  
>>> print x                # x changed or not?  
  
>>> # Repeat the test with:  
>>> x = "fortytwo"  
>>> x = [4, 2]  
>>> x = numpy.array([4, 2])
```

Argument passing - Experiments

```
>>> def twice2(a):  
...     a *= 2  
  
>>> x = 42  
>>> twice2(x)  
>>> print x                # x changed or not?  
  
>>> # Repeat the test with:  
>>> x = "fortytwo"  
>>> x = [4, 2]  
>>> x = numpy.array([4, 2])
```

Argument passing - Experiments

```
>>> def twice1(a):
```

```
...     a = 2 * a
```

```
>>> def twice2(a):
```

```
...     a *= 2
```

```
>>> x = [4, 2]
```

```
>>> print id(x)
```

```
>>> x *= 2
```

```
>>> print id(x)      # Changed in-place!
```

```
>>> x = 2 * x
```

```
>>> print id(x)      # New object!
```

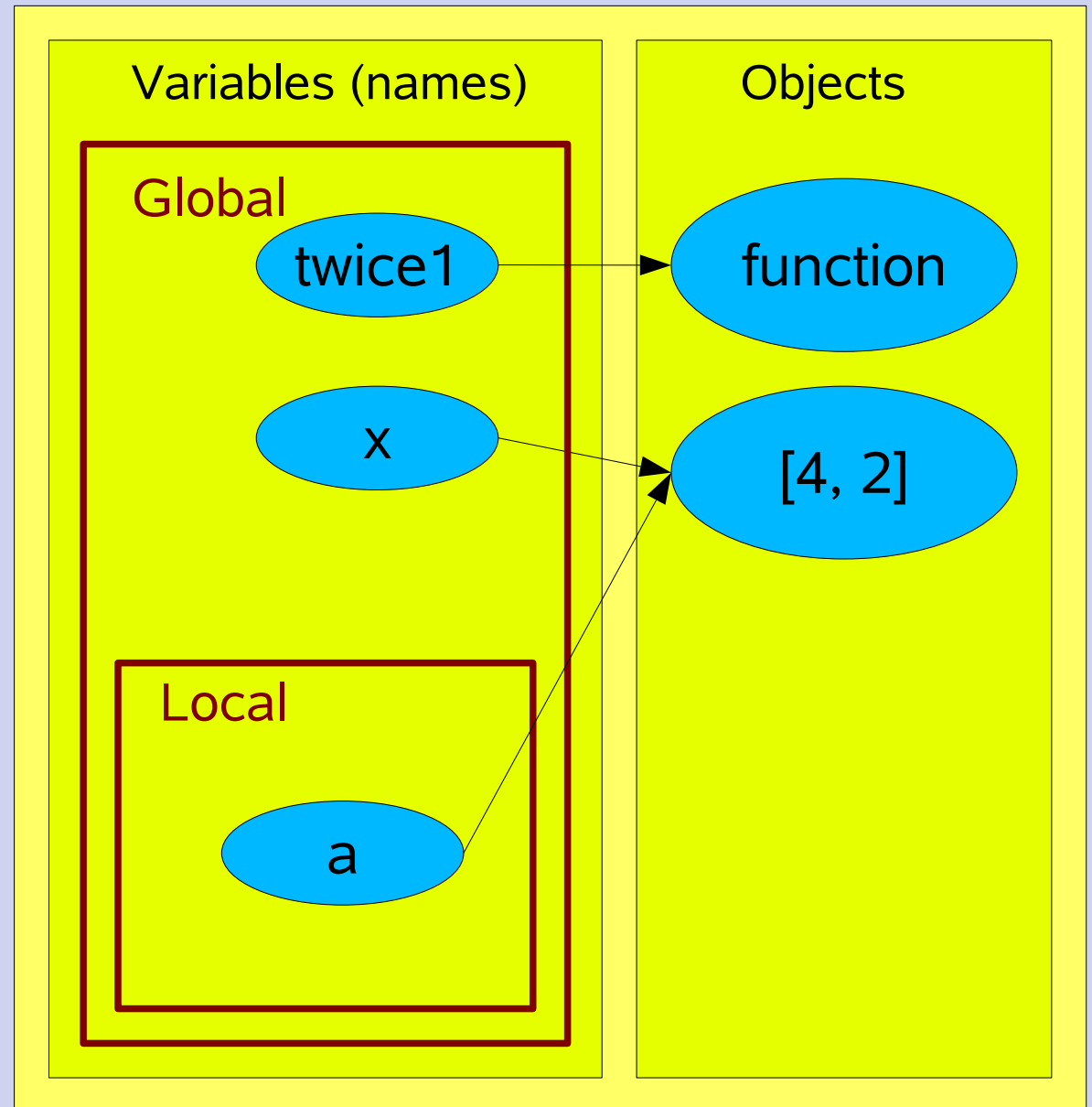
Argument passing

```
>>> def twice1(a):
```

```
...     a = 2 * a
```

```
>>> x = [4, 2]
```

```
>>> twice1(x)
```



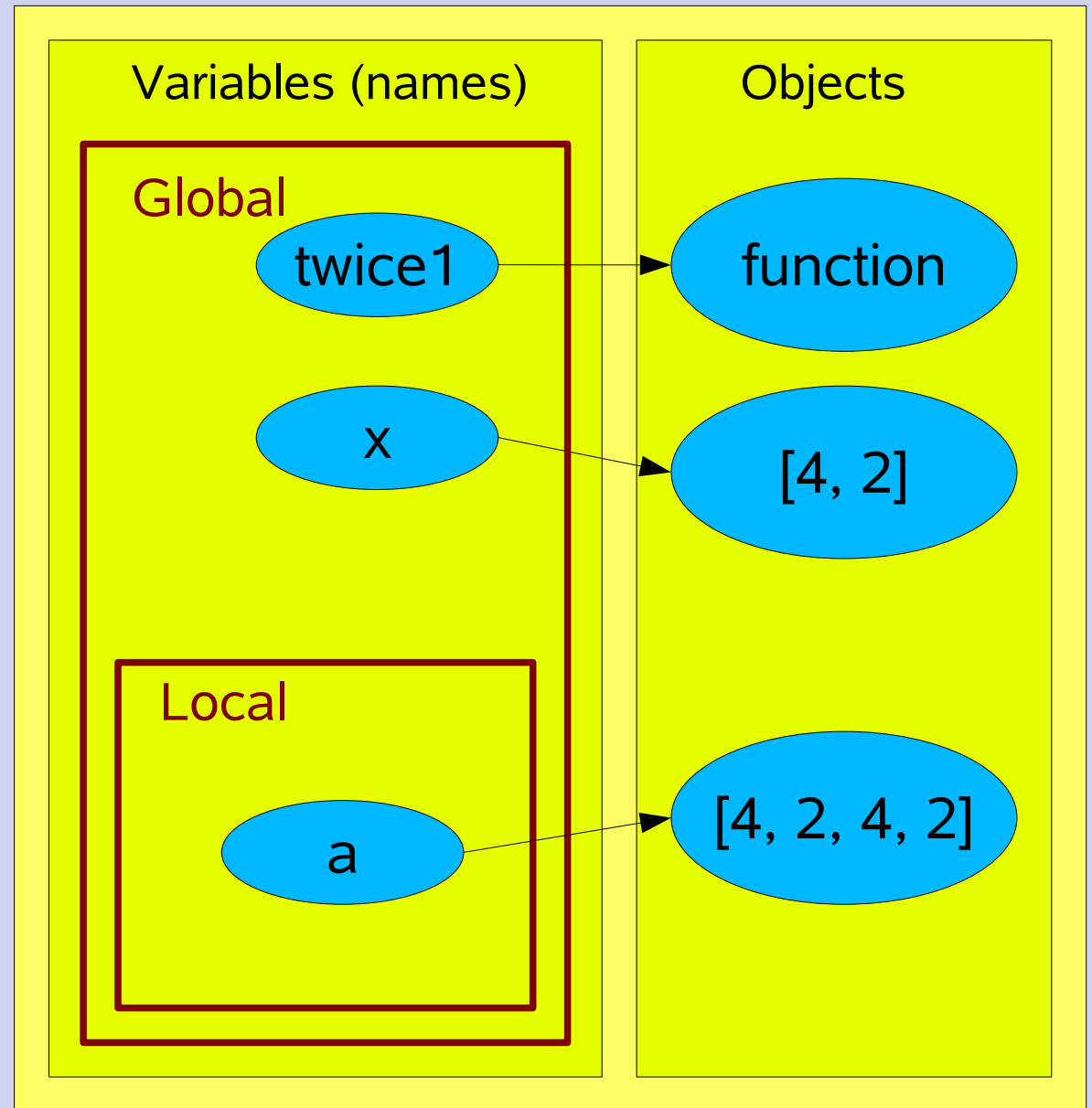
Argument passing

```
>>> def twice1(a):
```

```
...     a = 2 * a
```

```
>>> x = [4, 2]
```

```
>>> twice1(x)
```



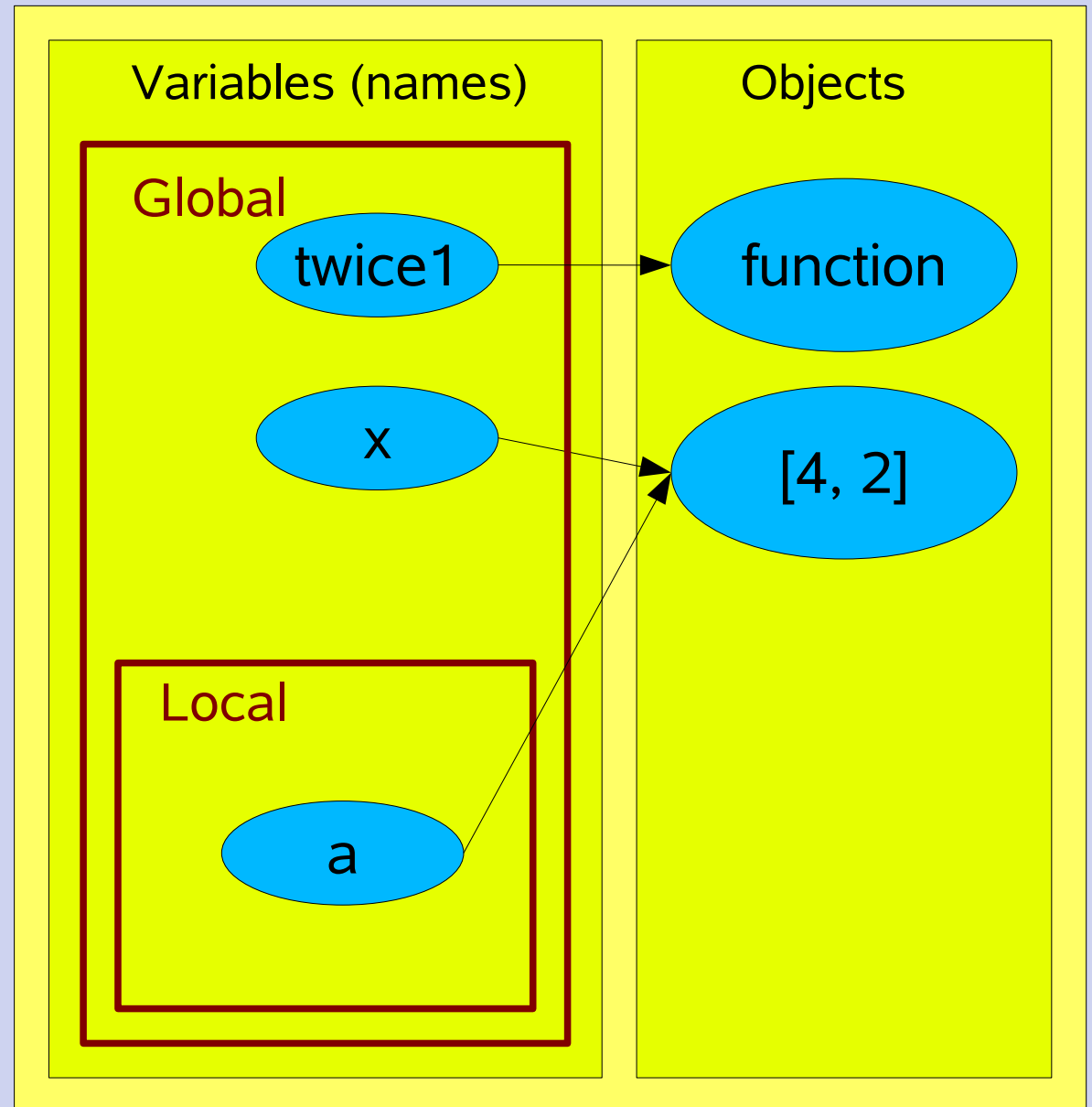
Argument passing

```
>>> def twice2(a):
```

```
...     a *= 2
```

```
>>> x = [4, 2]
```

```
>>> twice2(x)
```



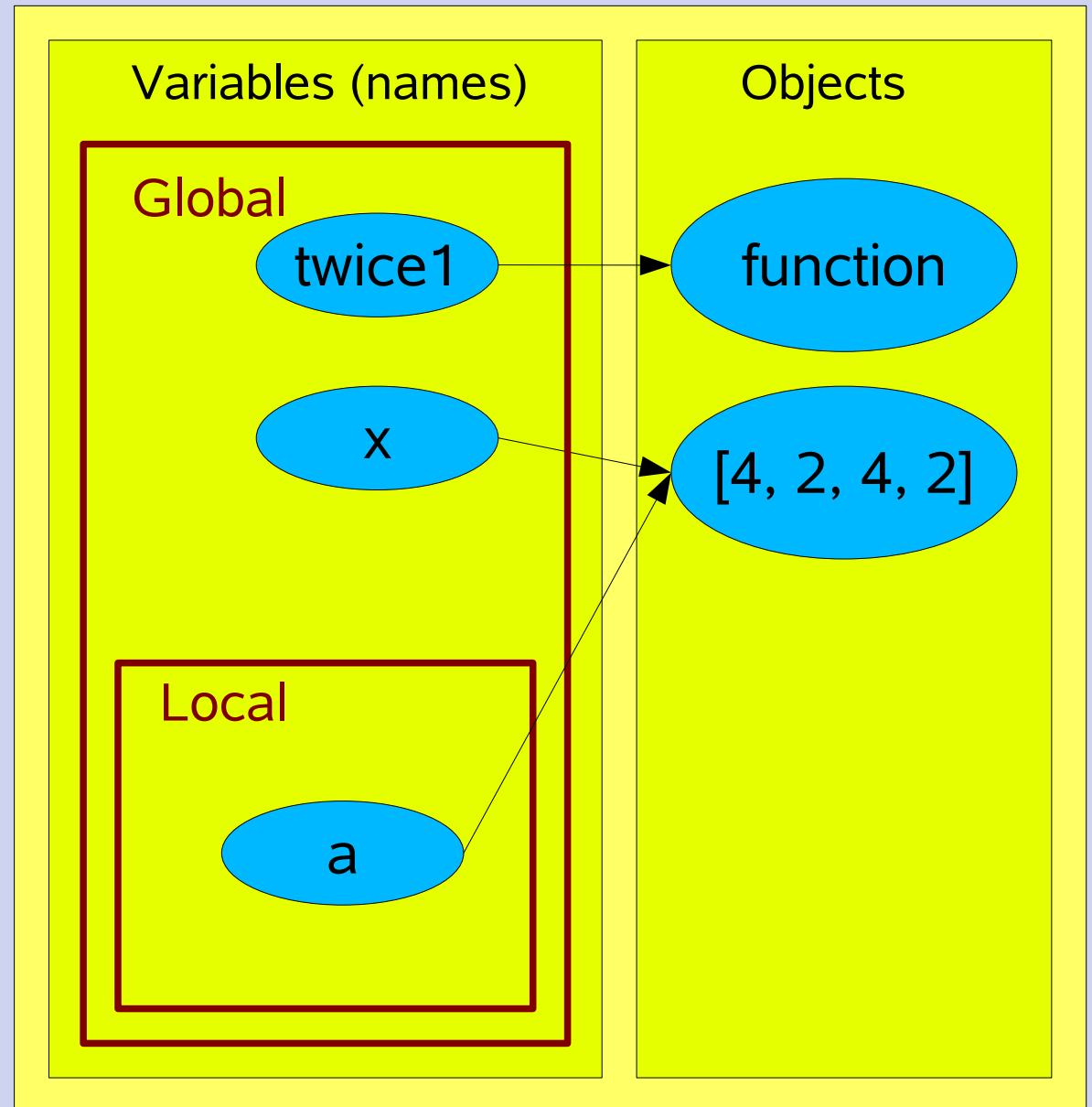
Argument passing

```
>>> def twice2(a):
```

```
...     a *= 2
```

```
>>> x = [4, 2]
```

```
>>> twice2(x)
```



Passing argument rules

- ***Immutable*** arguments act as if passed by value
- When changing ***mutable*** arguments ***in place*** inside the function, the object is changed outside the function too!

Reminder:

- Numbers, strings, tuples are immutable
- Lists, dictionaries, numpy.arrays are mutable

Safety measures

```
>>> def twice2(a):  
...     a *= 2           # a changed in place  
  
>>> # Pass copies:  
>>> x = [4, 2]  
>>> twice2(x[:])  
>>> print x           # x unchanged
```

Functions as objects

```
>>> def f(x="Hello world"):  
...     """Prints something"""  
...     print x  
  
>>> f()# Function call  
>>> f  # Function object, created by def statement  
  
>>> f.<TAB>  # Explore methods and attributes  
  
>>> print f.__doc__
```

Functions as objects - Consequences

```
>>> def f(x="Hello world"):  
...     print x  
  
>>> g = f # Assign new variable to function object  
>>> f("Printed from f")  
>>> g("Printed from g")  
  
>>> # Redefine function:  
>>> def f():  
...     print "f reloaded"  
  
>>> f()     # Redefined  
>>> g()     # Original
```

Functions as objects - Consequences

```
>>> # Conditional definition:
```

```
>>> x = True
```

```
>>> if x:
```

```
...     def f():
```

```
...         print "True f"
```

```
... else:
```

```
...     def f():
```

```
...         print "False f"
```

```
>>> f()
```

Functions as objects - Consequences

```
>>> # Dispatch dictionaries:
```

```
>>> def f():
```

```
...     print "f"
```

```
>>> def g():
```

```
...     print "g"
```

```
>>> disp = {"F": f, "G": g}
```

```
>>> disp["F"]()
```

Functions as objects - Consequences

```
>>> # Passing functions as arguments:
```

```
>>> def f():
```

```
...     print "f"
```

```
>>> def g(func):
```

```
...     print "g"
```

```
...     func()
```

```
>>> g(f)
```

Bonus 1: Nested functions

```
>>> import math
>>> def poisson(k, l):
...     """Poisson distribution"""
...     def nestFact(n):
...         # Used fact already earlier!
...         if n > 0:
...             return n * nestFact(n-1)
...         return 1
...     return (l**k * math.exp(-l)) / nestFact(k)

>>> poisson(2, 3)
>>> nestFact(2)          # Local to poisson!
```

Bonus 2: Arbitrary arguments

```
>>> # Arbitrary positional arguments:
```

```
>>> def f(*pargs):
```

```
...     print pargs      # tuple
```

```
>>> f("Spam", 42, [17, 4])
```

```
>>> # Arbitrary keyword arguments:
```

```
>>> def f(**kargs):
```

```
...     print kargs     # dictionary
```

```
>>> f(a="Spam", b=42, c=[17, 4])
```

Bonus 2: Arbitrary arguments

```
>>> # Combination:  
>>> def f(x, *pargs, **kargs):  
...     print x, pargs, kargs  
  
>>> f("X", "P0", "P1", karg0="K0", karg1="K1")
```

Bonus 2: Arbitrary arguments

```
>>> # Two simple examples:
```

```
>>> def minmax(*args):
```

```
...     l = list(args)
```

```
...     l.sort()
```

```
...     return l[0], l[-1]
```

```
>>> def fprintf(file, format, *args):
```

```
...     file.write(format % args)
```

Bonus 3: Argument unpacking

```
>>> # Complementary to arbitrary arguments:  
>>> limits = (1, 5)  
>>> range(*limits)
```

Modules

```
> # Get the example module file:
> cp ~rschaaf/lecture7.py .

>>> import lecture7      # Runs code in lecture7.py
>>>                       # and provides objects in
>>>                       # namespace lecture7:

>>> print lecture7.x      # Module attribute
>>> print lecture7.fact(10) # Module function
>>> print lecture.math.pi  # Nested module

>>> import lecture7      # Only run during first import
>>> reload(lecture7)     # But some tricky details
```

Modules as objects

```
>>> import lecture7      # Creates module object

>>> print lecture7

>>> lecture7.<TAB>      # Explore methods and attributes

>>> print lecture7.__doc__    # Doc string
>>> print lecture7.__name__   # "lecture7"
>>> print lecture7.__file__   # "lecture7.py" or
>>>                          # "lecture7.pyc"
```

Modules as objects - Consequences

```
>>> # Conditional imports:
```

```
>>> useLect7 = True
```

```
>>> if useLect7:
```

```
...     import lecture7
```

----- (Restart Python)

```
>>> # Local imports:
```

```
>>> def f():
```

```
...     import lecture7
```

```
...     print lecture7.poisson(2, 3)
```

----- (Restart Python)

```
>>> # Module name must make valid variable name:
```

```
>>> import if          # Fails even if if.py present
```

Module files as scripts

```
>>> # Imported from within Python:
```

```
>>> import lecture7      # __name__ = "lecture7"
```

```
>>>                      # Main program skipped
```

```
> # Run as script from shell:
```

```
> python lecture7.py    # __name__ = "__main__"
```

```
>                      # Main program executed
```

from statement

```
>>> x = "spam"
```

```
>>> from lecture7 import *
```

```
>>> # All attributes copied to global namespace:
```

```
>>> print x
```

```
>>> print poisson(2, 3)
```

----- (Restart Python)

```
>>> from lecture7 import poisson
```

```
>>> print x # Not imported
```

```
>>> print poisson(2, 3) # Ok
```

```
>>> print fact(10) # Not imported
```

----- (Restart Python)

import ... as ...

```
>>> import lecture7 as l7
```

```
>>> print l7.x
```

As word about code clearness

Use `from...import` and `import...as` with care. Both make your code harder to understand.

Do not sacrifice code clearness for some keystrokes!

In some cases, the use is acceptable:

- In interactive work (`import numpy as np`)
- If things are absolutely clear (e.g. all functions of an imported module obey a clear naming convention; `cfits_xyz`)
- `import.. as`: As last resort in case of name clashes between module names

Module search path

Modules are searched for in the following places:

- the current working directory (for interactive sessions)
- the directory of the top-level script file (for script files)
- the directories defined in PYTHONPATH
- Standard library directories
- Directories defined in .pth files

```
>>> # Get the complete module search path:  
>>> import sys  
>>> print sys.path
```

Using module packages

```
>>> import numpy
```

```
>>> numpy.random # Submodule
```

```
>>> numpy.random.randn() # Function in submodule
```

```
----- (Restart Python)
```

```
>>> import numpy.random # Import submodule only
```

```
>>> numpy.random.randn()
```

```
----- (Restart Python)
```

```
>>> from numpy import random # Alternative form
```

```
>>> random.randn()
```

```
----- (Restart Python)
```

```
>>> from numpy.random import * # Provisos from above
```

```
>>> randn() # apply here as well!
```

Exercise

Create a module `textstat` that contains the functions

- `openfile(filename, readwrite=False)`: opens the specified file (readonly or readwrite) and returns the open file object
- `isopen(file)`: returns True or False respectively
- `closefile(file)`: closes the file, if open
- `wordcount(file)`: returns the number of words in the file object
- `linecount(file)`: returns the number of lines in the file object
- `charcount(file)`: returns the number of characters in the file object

The module should contain a main program that uses these functions to print some statistics of `mobydick.txt`

Write a script that imports `textstat` and prints the same statistics of `mobydick.txt`