Lecture 3: Functions

Chirag Sakhuja

Overview

Recap

Basic Functions

Func-y Arguments

Scope

Functional Programming

Indexing

```
 0   1   2   3   4   5
name = 'Chirag'
-6  -5  -4  -3  -2  -1
```

String formatting

```python
str1 = 'Hello'
str2 = 'world'
'{}, {}!'.format(str1, str2)
# => 'Hello, world!'
'{0}, {1}, {0}'.format('first', 'second')
# => 'first, second, first'
'{:.2f}'.format(2.71828)
# => 2.72
```

Lists can contain anything, thanks duck typing!

```python
l = [1, 2, 3]
l = [1, 2, 'three']
l = [1, 2, [3, 4], [5]]

l.append('six')     # => [1, 2, [3, 4], [5], 'six']
```

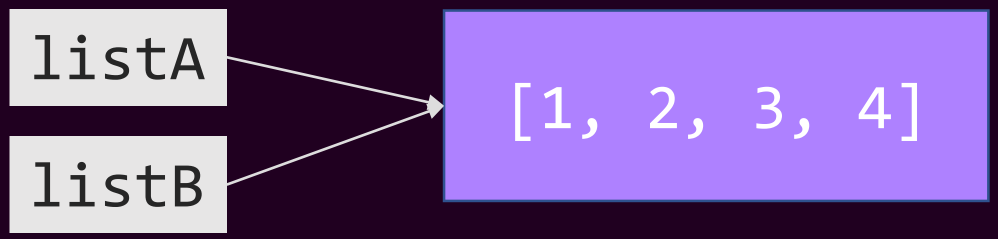
Lists denoted by square brackets

The in operator, again

```python
jagged = [[0], [1, 2], [3, 4, 5]]
0 in jagged                             # => False
[0] in jagged                           # => True
[1, 2] in jagged                        # => True
for row in jagged:
    print('{} ({})'.format(row, len(row)))
    # => [0] (1)
    # => [1, 2] (2)
    # => [3, 4, 5] (3)
```

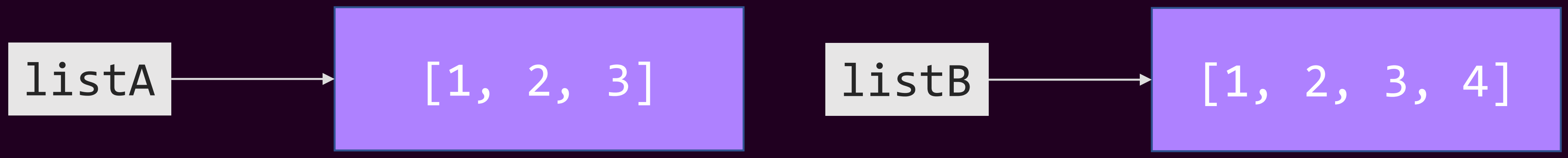
Digging a little deeper

```python
listA = [1, 2, 3]
listB = listA
listB.append(4)
print(listA)    # => [1, 2, 3, 4]
```



Digging a little deeper

```python
listA = [1, 2, 3]
listB = listA.copy()
listB.append(4)
print(listA)     # => [1, 2, 3]
```



Tuple packing/unpacking

```python
tup = 1, 2
```

Comma-separated r-values pack into a tuple

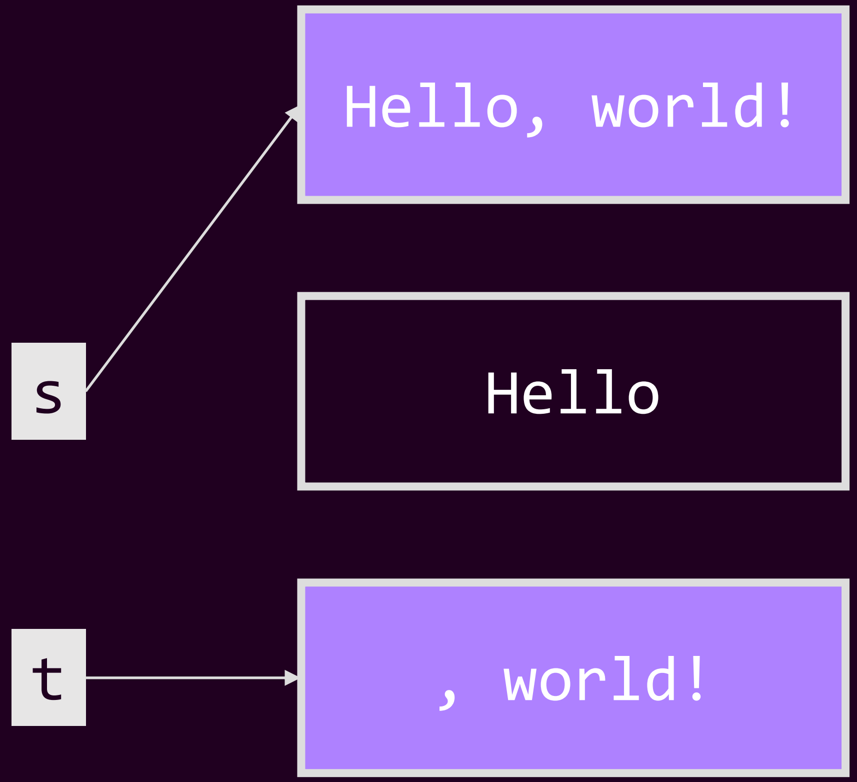
```python
a, b = tup
```

Comma-separated l-values unpack a tuple

```python
print(a)        # => 1
print(b)        # => 2
a, b = b, a
print(a)        # => 2
print(b)        # => 1
```

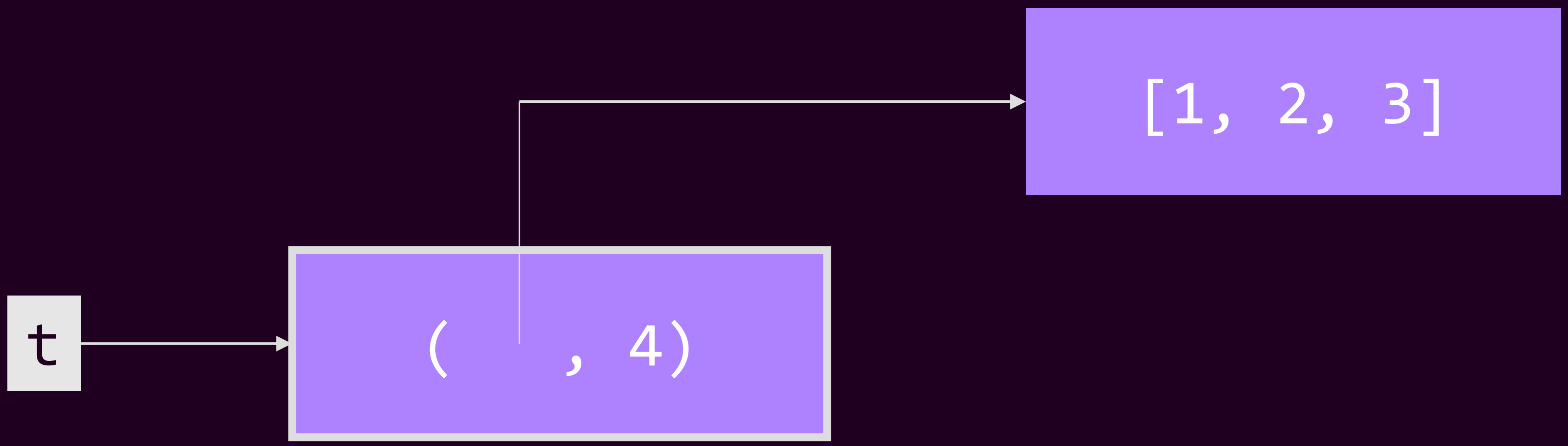
What does "immutable" mean?

```
s = 'Hello
t = ', world!'
s=s+t
```



What does "immutable" mean?

```python
tup = ([1, 2], 4)
tup[0].append(3)
tup[1] = 5      # => TypeError!
```



Sets can contain anything, thanks duck typing!

```python
s = {1, 2, 3}
s = {1, 2, 'three'}
s = set([1, 2, 3, 3])    # => {1, 2, 3}
s = set('Hello')         # => {'o', 'e', 'H', 'l'}
s[0]                     # => TypeError
```

Sets denoted by curly braces

Dicts

```python
a = dict(one=1, two=2)
b = {'one': 1, 'two': 2}
a == b                          # => True

empty = {}
```

Dict denoted by curly braces

Empty curly braces create dict, not set

Iterating over a dict

```python
grades = {'Chirag': [93, 87], 'Cassidy': [100, 94]}

for name, grade in grades.items():
    print('{}: {}'.format(name, grade))
    # => Chirag: [93, 87]
    # => Cassidy: [100, 94]
```

Overview

Recap

Basic Functions

Func-y Arguments

Scope

Functional Programming

"The Zen of Python" – Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

Basic function syntax

Start with : and then indent

```
def func_name(arg1, arg2):
    # do some stuff
    return arg1 + 1
```

Familiar return statements

```
func_name(1, 0)    # => 2
func_name(2, 0)    # => 3
```

Calls are just like C++

Building a more realistic Hello World

lib.py

```python
print('Hello, world!')
```

main.py

```python
import lib
```

CLI

```
> python3 lib.py
Hello, world!
> python3 main.py
Hello, world!
```

Shouldn't have printed anything!

Building a more realistic Hello World

lib.py

```python
if __name__ == '__main__':
    print('Hello, world!')
```

main.py

```python
import lib
```

CLI

```
> python3 lib.py
Hello, world!
> python3 main.py
```

No output, as expected

Structure of a runnable Python program

```python
def main():
    print('Hello, world!')

if __name__ == '__main__':
    main()
```

All functions return *something*

```
def do_nothing():
    return

type(do_nothing())     # => <class 'NoneType'>
```

What is *NoneType*?

- Something like a hybrid between `void` and NULL in C++
- A function that returns "nothing" returns *NoneType*
- Explicitly defined as None
- Evaluates to `False` in a conditional

Truthy and Falsy

Type	True when	False when
NoneType	Never	Always
bool	`True`	`False`
str	Non-empty	Empty
int	Not `0`	`0`
tuple	Non-empty	Empty
list	Non-empty	Empty
dict	Non-empty	Empty
set	Non-empty	Empty

Using Truthy and Falsy

Cannot call len on None

```
def fun(nums):
    if len(nums) == 0:
        print('empty')
    else:
        print('non-empty')

fun([1, 2, 3])      # => 'non-empty'
fun([])             # => 'empty'
fun(None)           # => TypeError!
```

Using Truthy and Falsy (better)

```python
def fun(nums):
    if not nums:
        print('empty')
    else:
        print('non-empty')

fun([1, 2, 3])     # => 'non-empty'
fun([])            # => 'empty'
fun(None)          # => 'empty'
```

Not really true

Using Truthy and Falsy (best)

```
def fun(nums):
    if nums is None:
        print('none')
    elif not nums:
        print('empty')
    else:
        print('non-empty')

fun([1, 2, 3])    # => 'non-empty'
fun([])           # => 'empty'
fun(None)         # => 'none'
```

is keyword checks if variables are pointing to the same memory block (just checks pointer addresses)

Works for None because None is cached

Duck typing, again and again

```python
def do_whatever(x):
    if   x % 3 == 1: return 'lalala'
    elif x % 3 == 2: return x

for i in range(3):
    print(do_whatever(i))    # => None
                             # => 'lalala'
                             # => 2
```

LBE: Sum of a list (without the library)

```
def sum(nums):
    if nums:
        res = 0
        for i in nums:
            res += i
        return res

sum([1, 2, 3])     # => 6
sum([])            # => None
sum(None)          # => None
```

Proper way to check if nums is non-empty and not None

On the else path, we don't return anything

LBE: Max of a list (without the library)

```python
def max(nums):
    if nums:
        res_idx = 0
        for idx, val in enumerate(nums):
            if val > nums[res_idx]:
                res_idx = idx
        return res_idx, nums[res_idx]

idx, val = max([3, 2, 1])     # => (0, 3)
idx, val = max([])            # => ???
```

LBE: Max of a list (without the library)

```
def max(nums):
    if nums:
        res_idx = 0
        for idx, val in enumerate(nums):
            if val > nums[res_idx]:
                res_idx = idx
        return res_idx, nums[res_idx]

idx, val = max([3, 2, 1])    # => (0, 3)
idx, val = max([])           # => TypeError!
```

Cannot unpack None

Overview

Recap

Basic Functions

Func-y Arguments

Scope

Functional Programming

Default arguments

```
def func(x, y=0):
    return x + y

func(1)         # => 1
func(1, 2)      # => 3
```

Keyword arguments

```
def func(x, y=0):
    return x + y

func(1)         # => 1
func(1, y=2)    # => 3
func(y=2, 1)    # => SyntaxError!
```

Positional arguments must come before keyword arguments

Keyword arguments

```python
def func(x, y=0):
    return x + y

func(1)         # => 1
func(1, z=2)    # => TypeError!
```

Only valid keyword arguments may be used
(unless using variadic keyword arguments)

LBE: Converting to integers

```
int('100')              # => 100
int('100', 16)          # => 256
int('100', base=8)      # => 64
```

Variadic positional arguments

Variable number of arguments are packed into a tuple

```
def func(*args):
    for x in args:
        print(x)


func(0)          # => 0
func(1, 2, 3)    # => 1
                       2
                       3
```

Variadic positional arguments

```
def func(*args):
    print(*args, sep=', ')
```

Unpack the tuple as individual arguments to print

```
print(0, sep=', ')           # => 0
print(1, 2, 3, sep=', ')     # => 1, 2, 3
func(0)                      # => 0
func(1, 2, 3)                # => 1, 2, 3
```

LBE: Arbitrary sized product

```python
def product(*nums, scale=1):
    res = scale
    for x in nums:
        res *= x
    return res

product(2, 3)               # => 6
nums = [2, 3, 4]
product(*nums)              # => 24
product(*nums, scale=2)     # => 48
```

Variadic keyword arguments

Excess keyword arguments are packed into dict

```
def cite(quote, **info):
    print('>', quote)
    print('-' * (len(quote) + 2))
    for k, v in info.items():
        print(k, v, sep=': ')

cite('Readability counts.',          # => > Readability counts.
     Title='The Zen of Python',      #    ---------------------
     Author='Tim Peters')            #    Title: The Zen of Python
                                     #    Author: Tim Peters
```

Variadic keyword arguments

```
def cite(quote, **info):
    print('>', quote)
    print('-' * (len(quote) + 2))
    for k, v in info.items():
        print(k, v, sep=': ')

info = {
    'Title': 'The Zen of Python',
    'Author': 'Tim Peters'
}
cite('Readability counts.', **info)  # => > Readability counts.
                                          ---------------------
                                          Title: The Zen of Python
                                          Author: Tim Peters
```

Unpack info dict into individual arguments

LBE: String formatting wrapper

```python
def printf(frm_str, *args, **kwargs):
    frm_str = 'DEBUG: ' + frm_str
    print(frm_str.format(*args, **kwargs))

printf('{}, {}!', 'Hello', 'world!')
# => 'DEBUG: Hello, world!'
printf('{0}, {1}, {0}', 'first', 'second')
# => 'DEBUG: first, second, first'
printf('{} {e:.2f}', 3.14, e=2.71828)
# => 'DEBUG: 3.14 2.72'
```

Overview

Recap

Basic Functions

Func-y Arguments

Scope

Functional Programming

Scope in C++

```
std::string foo(bool test) {
    std::string msg = "";
    if(test) {
        int x;
        msg = "success!";
    } else {
        msg = "failure :(";
    }
    return msg;
}
```

Need to declare `msg` outside of if statement so it's accessible for return

Braces create a new scope, so x is only accessible inside if statement

Scope in Python

```
def foo(test):
    if test:
        msg = 'success!'
    else:
        msg = 'failure :('
    return msg
```

Only functions (and classes…) create new scope, and everything declared anywhere in the function is accessible

Querying scope

```
x = 1
def foo(y):
    z = 3
    print(locals())
    print(globals())

foo(2)     # => {'z': 3, 'y': 2}
           # => { ..., 'x': 1, ...}
```

... because of lots of built-in things

Variables bind to nearest scope

x in local and global scope

```
x = 1
def foo(x):
    z = 3
    print(locals())
    print(globals())

foo(2)     # => {'z': 3, 'x': 2}
           # => { ..., 'x': 1, ...}
```

It picks the nearest scope for locals

But globals are unchanged

Reading a file

```
f = open('input.txt', 'r')
f.readlines()    # => ['From\n', 'the\n', 'file\n']
# some stuff
f.close()
```

Reading a file

```
f = open('input.txt', 'r')
f.readlines()     # => ['From\n', 'the\n', 'file\n']
x = 1 / 0         # => ZeroDivisionError!
f.close()
```

Program crashes before releasing the file
The OS won't be happy about that!

LBE: Reading a file

If there's an exception in the with block, f is properly destroyed

```
with open('input.txt', 'r') as f:
    lines = f.readlines()
print(lines)    # => ['From\n', 'the\n', 'file\n']
```

lines is still in scope

Overview

Recap

Basic Functions

Func-y Arguments

Scope

Functional Programming

Programming paradigms [1]

- Procedural (subcategory of Imperative): Programming with an explicit sequence of commands that updates state
  - Example: C
- Declarative: Programming by specifying the result you want, not how you get it
  - Examples: Prolog, SQL
- Object-Oriented: Programming by defining objects that send messages to each other over well-specified interfaces
  - Example: Java

Programming paradigms [1]

- Functional: Programming with function calls that avoid any global state
  - Example: Haskell
- Multi-Paradigm: Programming with multiple paradigms combined freely
  - Examples: C++, Python

Functional programming

- Program is composed of functions that do not have side effects
  - Do not modify global state
  - Do not perform I/O (e.g. printing on the screen is a side effect)
- Derived from lambda calculus
  - Easy to apply formal techniques
- Easy to reason about and test functions
- More conducive to being parallelized

type(example) != FunctionalProgramming

```
nums = [3, 4, 1, 5, 2]
nums.sort()
```

Side effect: nums is changed

```
print(nums)    # => [1, 2, 3, 4, 5]
```

```
type(example) == FunctionalProgramming

nums = [3, 4, 1, 5, 2]
sorted_nums = sorted(nums)

print(nums)               # => [3, 4, 1, 5, 2]
print(sorted_nums)        # => [1, 2, 3, 4, 5]
```

No side effects

Higher order functions [2]

A function that does at least one of the following

- Takes one or more functions as arguments
- Returns a function

Functions are objects

```
def foo(x):
    print('foo with {}'.format(x))

foo(1)          # => 'foo with 1'
bar = foo
bar(2)          # => 'foo with 2'
```

Functions are objects

```
def foo(x):
    print('foo with {}'.format(x))

type(foo) # => <class 'function'>
foo(foo)  # => foo with <function foo at 0x7f16530ede18>
```

Functions are objects

```python
def add(x, y):
    return x + y

def operate(fun, x, y):
    return fun(x, y)

operate(add, 1, 2)    # => 3
```

Function factory

```python
def factory(x):
    def helper(y):
        return x + y
    return helper

add1 = factory(1)
add2 = factory(2)
add1(10)         # => 11
add2(10)         # => 12
factory(3)(10)   # => 13
```

Function factory, detailed

Scope A

Scope B

```
def factory(x):
    def helper(y):
        return x + y
    return helper
```

```
add1 = factory(1)
add2 = factory(2)
add1(10)          # => 11
add2(10)          # => 12
factory(3)(10)    # => 13
```

Scope B includes the values from Scope A

- Mutable types are copied by reference
- Immutable types are copied by value

Imagine everything in Scope A being copied into Scope B using the assignment operator

LBE: Simple calculator paradigm

```python
def add(x, y): return x + y
def sub(x, y): return x - y

def operate(op, x, y):
    if   op == '+': func = add
    elif op == '-': func = sub
    return func(x, y)

operate('-', 1, 2)    # => -1
```

LBE: Logging functions (before)

```python
def printMsg(msg_type, msg):
    print('{}: {}'.format(msg_type, msg))

def printErrMsg(msg):
    printMsg('Error', msg)

def printWarnMsg(msg):
    printMsg('Warn', msg)

printErrMsg('no file!')        # => 'Error: no file!'
printWarnMsg('found typo')     # => 'Warn: found typo'
```

LBE: Logging functions (after)

```
def printMsg(msg_type):
    def printMsgHelper(msg):
        print('{}: {}'.format(msg_type, msg))
    return printMsgHelper

printErrMsg = printMsg('Error')
printWarnMsg = printMsg('Warn')

printErrMsg('no file!')       # => 'Error: no file!'
printWarnMsg('found typo')    # => 'Warn: found typo'
```

LBE: Tail recursive sum

```
def sum(nums):
    def helper(nums, res):
        if nums:
            num = nums.pop()
            return helper(nums, res + num)
        return res
    return helper(nums, 0)

sum([1, 2, 3])          # => 6
helper([1, 2, 3], 0)    # => NameError!
```

LBE: Tail recursive sum (corrected)

```
def sum(nums):
    def helper(nums, res):
        if nums:
            num = nums.pop()
            return helper(nums, res + num)
        return res
    return helper(nums.copy(), 0)

sum([1, 2, 3])          # => 6
helper([1, 2, 3], 0)    # => NameError!
```

Don't forget the Zen

```python
def sum(nums, idx=0, res=0):
    if idx < len(nums):
        num = nums[idx]
        return sum(nums, idx + 1, res + num)
    return res

sum([1, 2, 3])          # => 6
```

LBE: Closures

```python
def create_counter():
    count = 0
    def helper():
        count += 1
        return count
    return helper

counter = create_counter()
counter()                    # => UnboundLocalError!
counter()
```

LBE: Closures (corrected)

```
def create_counter():
    count = [0]
    def helper():
        count[0] += 1
        return count[0]
    return helper

counter1 = create_counter()
counter1()                        # => 1
counter1()                        # => 2
counter2 = create_counter()
counter2()                        # => 1
```

No longer functional; `helper` modifies something outside its scope

Called a closure because it encompasses all state

Key insights

- You have lots and lots of options
- Learning different design paradigms takes time
  - Eventually you'll know which is best for the job
- Don't get bogged down by everything that's available, and use what you're comfortable with…and then a little bit more

References

[1] http://cs.lmu.edu/~ray/notes/paradigms/

[2] https://en.wikipedia.org/wiki/Higher-order_function

[3] http://stanfordpython.com/