User Defined Functions, Text Data, Output Formating

- User-Defined Functions
- Assignments Revisited and Parameter Passing
- Strings, revisited
- Formatted output

Defining new functions

A few built-in functions we have seen:

- `abs()`, `max()`, `len()`, `sum()`, `print()`

New functions can be defined using `def`

`def`: function definition keyword

`f`: name of function

`x`: variable name for input argument

```
def f(x):
    res = x**2 + 10
    return res
```

`return`: specifies function output

```
>>> abs(-9)
9
>>> max(2, 4)
4
>>> lst = [2,3,4,5]
>>> len(lst)
4
>>> sum(lst)
14
>>> print()

>>> def f(x):
        res = 2*x + 10
        return x**2 + 10

>>> f(1)
11
>>> f(3)
19
>>> f(0)
10
```

`print()` versus `return`

```
def f(x):
    res = x**2 + 10
    return res
```

```
>>> f(2)
14
>>> 2*f(2)
28
```

Function returns value of `res` which can then be used in an expression

```
def f(x):
    res = x**2 + 10
    print(res)
```

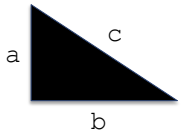
```
>>> f(2)
14
>>> 2*f(2)
14
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in
<module>
    2*f(2)
TypeError: unsupported operand
type(s) for *: 'int' and
'NoneType'
```

Function prints value of `res` but does not return anything

Defining new functions

The general format of a function definition is

```
def <function name> (<0 or more variables>):
    <indented function body>
```



Let's develop function `hyp()` that:

- Takes two numbers as input (side lengths a and b of above right triangle)
- Returns the length of the hypotenuse c

```
>>> hyp(3,4)
5.0
>>>
```

```
import math
def hyp(a, b):
    return math.sqrt(a**2 + b**2)
```

Exercise

Write function `hello()` that:

- takes a name (i.e., a string) as input
- prints a personalized welcome message

Note that the function does not return anything

```
>>> hello('Julie')
Welcome, Julie, to the world of Python.
>>>
```

```
def hello(name):
    line = 'Welcome, ' + name + ', to the world of Python.'
    print(line)
```


Exercise

Write function `rng()` that:

- takes a list of numbers as input
- returns the range of the numbers in the list

The range is the difference between the largest and smallest number in the list

```
>>> rng([4, 0, 1, -2])
6
>>>
```

```
def rng(lst):
    res = max(lst) - min(lst)
    return res
```

Comments and docstrings

Python programs should be documented

- So the developer who writes/maintains the code understands it
- So the user knows what the program does

Comments

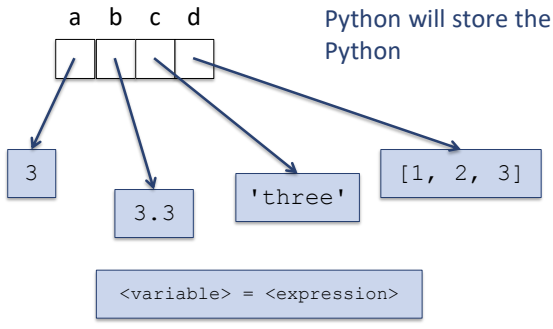
```
def f(x):
    res = x**2 + 10    # compute result
    return res         # and return it
```

Docstring

```
def f(x):
    'returns x**2 + 10'
    res = x**2 + 10    # compute result
    return res         # and return it
```

```
>>> help(f)
Help on function f in module
__main__:

f(x)

>>> def f(x):
        'returns x**2 + 10'
        res = x**2 + 10
        return res

>>> help(f)
Help on function f in module
__main__:

f(x)
    returns x**2 + 10

>>>
```


Assignment statement: a second look

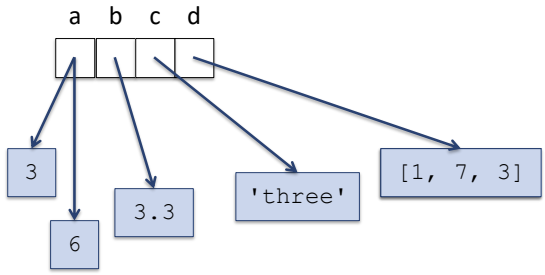


A variable does not exist before it is assigned
Python will store the name in a table maintained by Python

1. `<expression>` is evaluated and its value put into an object of appropriate type
2. The object is assigned name `<variable>`

```
>>> a
Traceback (most recent call
last):
  File "<pyshell#66>", line
1, in <module>
    a
NameError: name 'a' is not
defined
>>> a = 3
>>> b = 2 + 1.3
>>> c = 'three'
>>> d = [1, 2] + [3]
```

Mutable and immutable types




```
>>> a
3
>>> a = 6
>>> a
6
>>> d
[1, 2, 3]
>>> d[1] = 7
>>> d
[1, 7, 3]
```

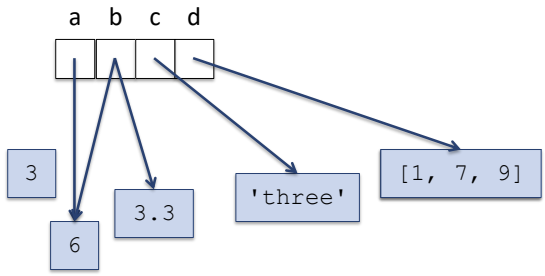
The object (3) referred to by variable `a` does not change; instead, `a` refers to a new object (6)

- Integers are immutable

The object ([1, 2, 3]) referred to by `d` changes

- Lists are mutable

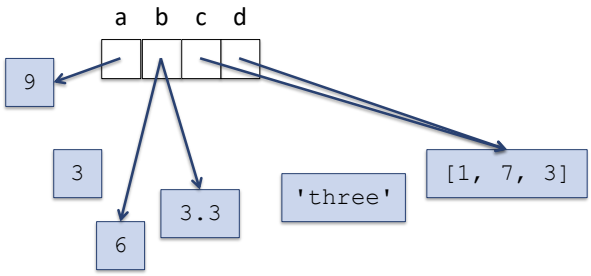
Assignment and mutability




```
>>> a
6
>>> b
3.3
>>> b = a
>>> b
6
```

`a` and `b` refer to the same integer object

Assignment and mutability



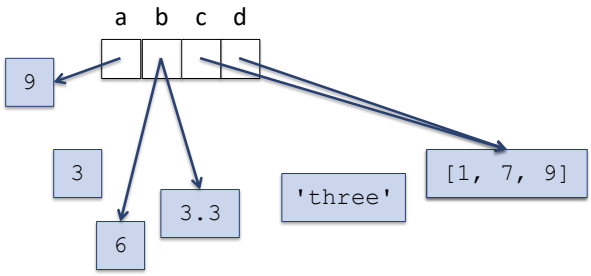

```
>>> a
6
>>> b
3.3
>>> b = a
>>> b
6
>>> a = 9
>>> b
6
>>> c = d
>>> c
[1, 7, 3]
```

a now refers to a new object (9); b still refers to the old object (6)

- Because integers are immutable, a change to a does not affect the value of b

c and d refer to the same list object

Assignment and mutability



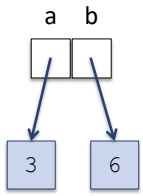

```
>>> a
6
>>> b
3.3
>>> b = a
>>> b
6
>>> a = 9
>>> b
6
>>> c = d
>>> c
[1, 7, 3]
>>> d[2] = 9
>>> c
[1, 7, 9]
```

The list that c refers to changes; d refers to the same list object, so it changes too

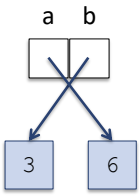
- Because lists are mutable, a change to d affects c

Swapping values

Have:



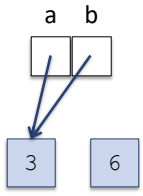
Want:



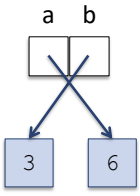

```
>>> a
3
>>> b
6
```


Swapping values

Have:



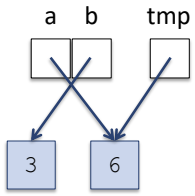
Want:



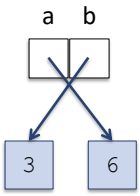

```
>>> a
3
>>> b
6
>>> b = a
```

Swapping values

Have:



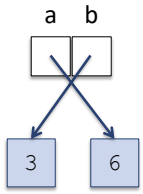
Want:



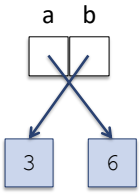

```
>>> a
3
>>> b
6
>>> tmp = b
>>> b = a
>>> a = tmp
```


Swapping values

Have:

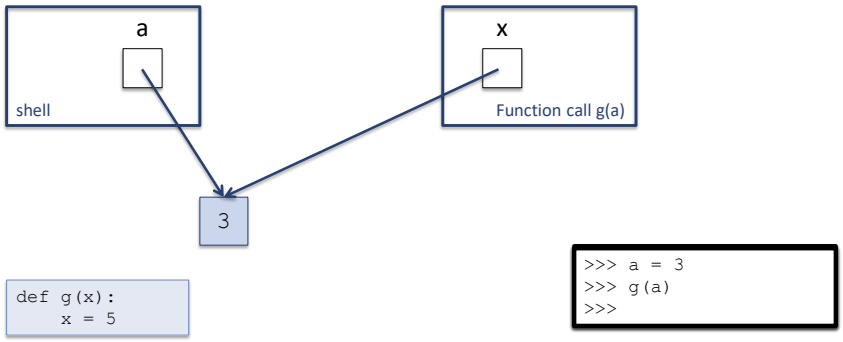


Want:




```
>>> a
3
>>> b
6
>>> a, b = b, a
```

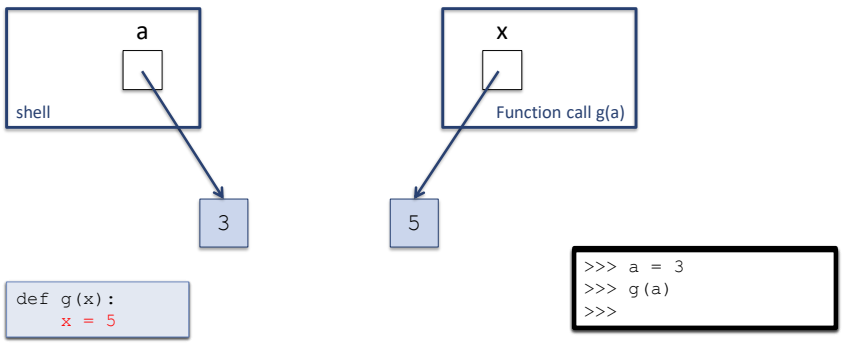
Immutable parameter passing



Variable `x` inside `g()` refers to the object `a` refers to

As if we executed x = a

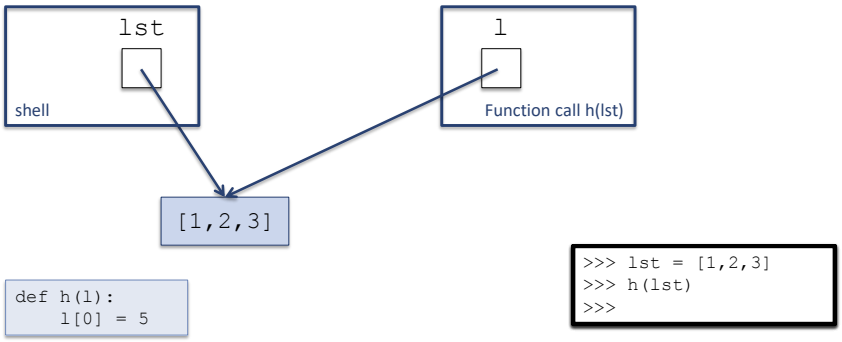
Immutable parameter passing



Function `g()` did not, and cannot, modify the value of `a` in the interactive shell.

This is because `a` refers to an immutable object.

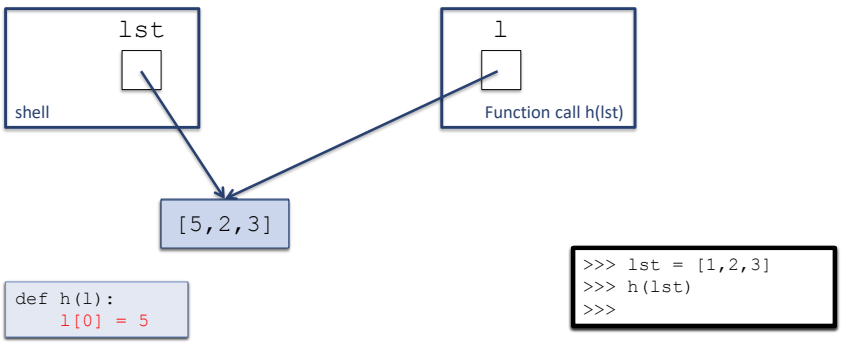
Mutable parameter passing



Variable `l` inside `h()` refers to the object `lst` refers to

As if we executed `l = lst`

Mutable parameter passing



Function `h()` did modify the value of `lst` in the interactive shell.

This is because `lst` and `l` refer to an mutable object.

Exercise

Write function `swapFS()` that:

- takes a list as input
- swaps the first and second element of the list, but only if the list has at least two elements

The function does not return anything

```
def swapFS(lst):
    if len(lst) > 1:
        lst[0], lst[1] = lst[1], lst[0]
```

```
>>> mylst = ['one', 'two', 'three']
>>> swapFS(mylst)
>>> mylst
['two', 'one', 'three']
>>> mylst = ['one']
>>> swapFS(mylst)
>>> mylst
['one']
>>>
```


String representations

A string value is represented as a sequence of characters delimited by quotes

Quotes can be single (') or double (")

What if ' or " is one of the string characters?

What if the string includes both ' and "?

Escape sequence \' or \" is used to indicate that a quote is not the string delimiter but is part of the string value

Function `print()` interprets the escape sequence

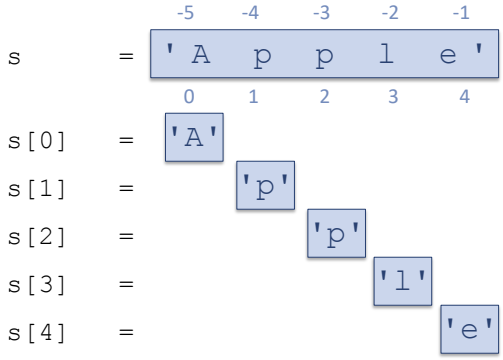
Another example:

- `\n` is an escape sequence that represents a new line

```
>>> excuse = 'I am sick'
>>> excuse = "I am sick"
>>> excuse = 'I'm sick'
SyntaxError: invalid syntax
>>> excuse = "I'm sick"
>>> excuse = "I'm "sick""
SyntaxError: invalid syntax
>>> excuse = 'I'm "sick"'
SyntaxError: invalid syntax
>>> excuse = 'I\'m "sick"'
>>> excuse
'I\'m "sick"'
>>> print(excuse)
I'm "sick”
>>> excuse = 'I\'m ...\n... "sick"'
>>> excuse
'I\'m ...\n... "sick"'
>>> print(excuse)
I'm ...
... "sick"
```

Indexing operator, revisited

The indexing operator returns the character at index i (as a single character string).




```
>>> s = 'Apple'
>>> s[0]
'A'
>>> s[1]
'p'
>>> s[4]
'e'
```

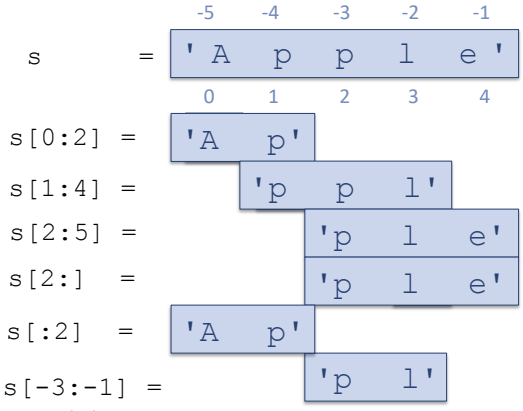

Indexing operator, revisited

The indexing operator can also be used to obtain a slice of a string

`s[i:j]` : the slice of `s` starting at index `i` and ending before index `j`

`s[i:]` : the slice of `s` starting at index `i`

`s[:j]` : the slice of `s` ending before index `j`




```
>>> s = 'Apple'
>>> s[0:2]
'Ap'
>>> s[1:4]
'ppl'
>>> s[2:5]
'ple'
>>> s[2:]
'ple'
>>> s[:2]
'Ap'
>>> s[-3:-1]
'pl'
```

Exercise

The indexing operator can also be used to obtain slices of a list as well. Let list `lst` refer to list

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Write Python expressions using list `lst` and the indexing operator that evaluate to:

a) `['a', 'b', 'c', 'd']`
b) `['d', 'e', 'f']`
c) `['d']`
d) `['f', 'g']`
e) `['d', 'e', 'f', 'g', 'h']`
f) `['f', 'g', 'h']`

```
>>> lst[:4]
['a', 'b', 'c', 'd']
>>> lst[3:6]
['d', 'e', 'f']
>>> lst[3:4]
['d']
>>> lst[-3:-1]
['f', 'g']
>>> lst[3:]
['d', 'e', 'f', 'g', 'h']
>>> lst[-3:]
['f', 'g', 'h']
```


String methods

Strings are immutable; none of the string methods modify string `link`

```
>>> link = 'http://www.main.com/smith/index.html'
>>> link[:4]
'http'
>>> link[:4].upper()
'HTTP'
>>> link.find('smith')
20
>>> link[20:25]
'smith'
>>> link[20:25].capitalize()
'Smith'
>>> link.replace('smith', 'ferreira')
'http://www.main.com/ferreira/index.html'
>>> link
'http://www.main.com/smith/index.html'
>>> new = link.replace('smith', 'ferreira')
>>> new
'http://www.main.com/ferreira/index.html'
>>> link.count('/')
4
>>> link.split('/')
['http:', '', 'www.main.com', 'smith', 'index.html']
```

String methods

Strings are immutable; none of the string methods modify string `s`

Usage	Explanation
`s.capitalize()`	returns a copy of `s` with first character capitalized
`s.count(target)`	returns the number of occurences of `target` in `s`
`s.find(target)`	returns the index of the first occurrence of `target` in `s`
`s.lower()`	returns lowercase copy of `s`
`s.replace(old, new)`	returns copy of `s` with every occurrence of `old` replaced with `new`
`s.split(sep)`	returns list of substrings of `s`, delimited by `sep`
`s.strip()`	returns copy of `s` without leading and trailing whitespace
`s.upper()`	returns lowercase copy of `s`

Exercise

```
>>> events = '9/13 2:30 PM\n9/14 11:15 AM\n9/14 1:00 PM\n9/15 9:00 AM'
>>> print(events)
9/13 2:30 PM
9/14 11:15 AM
9/14 1:00 PM
9/15 9:00 AM
```

String `events` describes the schedule of 4 events spread across 3 days

Write expressions that compute:

a) the number of events on 9/14
b) the index of the substring describing the 1st event on 9/14
c) the index just past the substring describing the last event on 9/14
d) the list of substrings describing the events on 9/14

```
>>> events.count('9/14')
2
>>> events.find('9/14')
13
>>> events.find('9/15')
40
>>> events[13:40]
'9/14 11:15 AM\n9/14 1:00 PM\n'
>>> lst = events[13:40].strip().split('\n')
>>> lst
['9/14 11:15 AM', '9/14 1:00 PM']
>>>
```

String methods

Suppose we need to pick up the date and time components of string `event`

Puntuation makes it difficult to use method `split()`

```
>>> event = "Tuesday, Feb 29, 2012 -- 3:35 PM"
>>> table = str.maketrans(':,-', 3*' ')
>>> event.translate(table)
'Tuesday  Feb 29  2012    3 35 PM'
>>> event.translate(table).split()
['Tuesday', 'Feb', '29', '2012', '3', '35', 'PM']
>>>
```

Solution: replace punctuation with blank spaces

Usage	Explanation
`str.maketrans(old, new)`	returns a table mapping characters in string `old` to characters in string `new`
`s.translate(table)`	returns a copy of `s` in which the original characters are replaced using the mapping described by `table`

Built-in function `print()`, revisited

their string representation

Function `print()` takes 0 or more arguments and prints ~~them~~ in the shell

```
>>> prod = 'morels'
>>> cost = 139
>>> wght = 1/2
>>> total = cost * wght
>>> print(prod, cost, wght, total)
morels 139 0.5 69.5
>>> print(prod, cost, wght, total, sep='; ')
morels; 139; 0.5; 69.5
>>> print(prod, cost, wght, total, sep=':::')
morels:::139:::0.5:::69.5
>>>
```

A blank space separator is printed between the arguments

The `sep` argument allows for customized separators

Built-in function print(), revisited

Function `print()` prints, by default, a newline character after printing its arguments

```
>>> pets = ['boa', 'cat', 'dog']
>>> for pet in pets:
        print(pet)


boa
cat
dog
>>> for pet in pets:
        print(pet, end=', ')


boa, cat, dog,
>>> for pet in pets:
        print(pet, end='!!! ')


boa!!! cat!!! dog!!!
>>>
```

The `end` argument allows for customized end characters

General output formatting

Suppose we have

```
>>> weekday = 'Wednesday'
>>> month = 'March'
>>> day = 10
>>> year = 2010
>>> hour = 11
>>> minute = 45
>>> second = 33
>>> print(hour+':'+minute+':'+second)
Traceback (most recent call last):
  File "<pyshell#113>", line 1, in <module>
    print(hour+':'+minute+':'+second)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> print(str(hour)+':'+str(minute)+':'+str(second))
11:45:33
>>> print('{}:{}:{}'.format(hour, minute, second))
11:45:33
```

and we want to print `Wednesday, March 10, 2010 at 11:45:33`

Method `format()` of class `str`

```
>>> day = 'Wednesday'
>>> month = 'March'
>>> weekday = 'Wednesday'
>>> month = 'March'
>>> day = 10
>>> year = 2010
>>> year = 2012
>>> hour = 11
>>> minute = 45
>>> second = 33
>>> print('{}:{}:{}'.format(hour, minute, second))
11:45:33
>>> print('{}, {} {}, {} at {}:{}:{}'.format(weekday, month,
day, year, hour, minute, second))
Wednesday, March 10, 2012 at 11:45:33
```

format string

```
print('{}:{}:{}'.format(hour, minute, second))
```

placeholders

Specifying field width

The `format()` method can be used to line up data in columns

Numbers are aligned to the right

```
>>> for i in range(1,8):
        print(i, i**2, 2**i)


1 1 2
2 4 4
3 9 8
4 16 16
5 25 32
6 36 64
7 49 128
>>> for i in range(1, 8):
        print('{} {:2} {:3}'.format(i, i**2,
2**i))


1  1   2
2  4   4
3  9   8
4 16  16
5 25  32
6 36  64
7 49 128
>>>
```

reserves 3 spaces for `2**i`

reserves 2 spaces for `i**2`

plus a blank space between the columns

Specifying field width

The `format()` method can be used to line up data in columns

Numbers are aligned to the right

Strings are aligned to the left

```
>>> lst = ['Alan Turing', 'Ken Thompson', 'Vint Cerf']
>>> for name in lst:
        fl = name.split()
        print(fl[0], fl[1])


Alan Turing
Ken Thompson
Vint Cerf
>>> for name in lst:
        fl = name.split()
        print('{:5} {:10}'.format(fl[0], fl[1]))


Alan  Turing
Ken   Thompson
Vint  Cerf
>>>
```


Output format type

Inside the curly braces of a placeholder, we can specify the field width, the type of the output, and the decimal precision

Type	Explanation
b	binary
c	character
d	decimal
X	hexadecimal
e	scientific
f	fixed-point

```
>>> n = 10
>>> '{:b}'.format(n)
'1010'
>>> '{:c}'.format(n)
'\n'
>>> '{:d}'.format(n)
'10'
>>> '{:X}'.format(n)
'A'
>>> '{:e}'.format(n)
'1.000000e+01'
>>> '{:7.2f}'.format(n)
'  10.00'
>>>
```