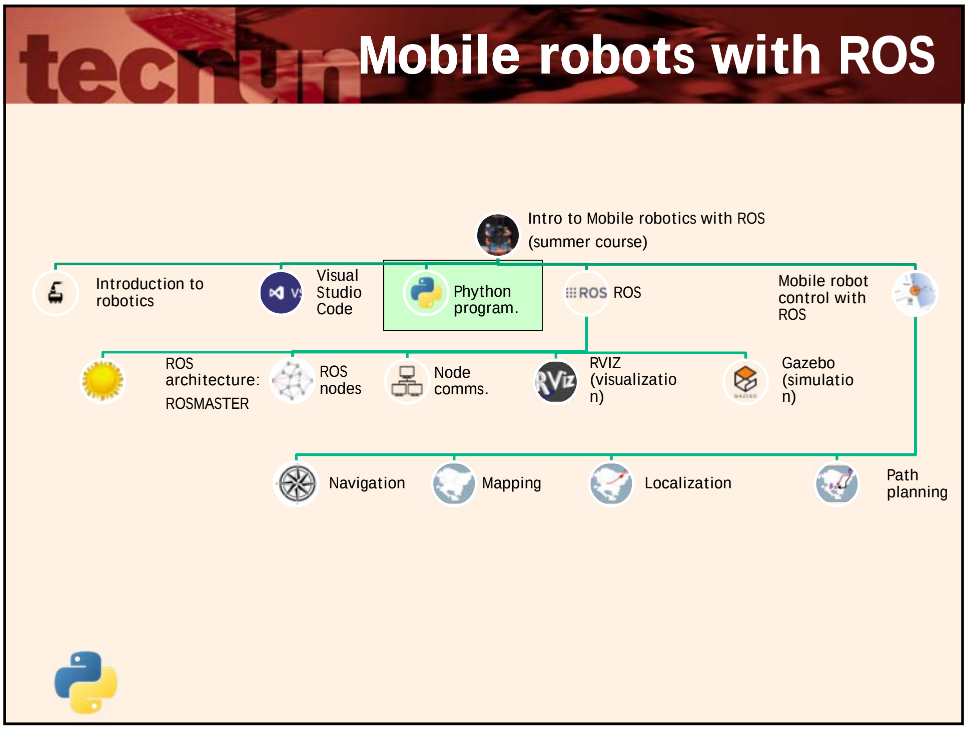
Python Tutorial
Release 2.7.15

Guido van Rossum
and the Python development team



Mobile robots with ROS

Intro to Mobile robotics with ROS
(summer course)

Introduction to robotics

Visual Studio Code

Phython program.

ROS

Mobile robot control with ROS

ROS architecture:
ROSMASTER

ROS nodes

Node comms.

RVIZ (visualizatio n)

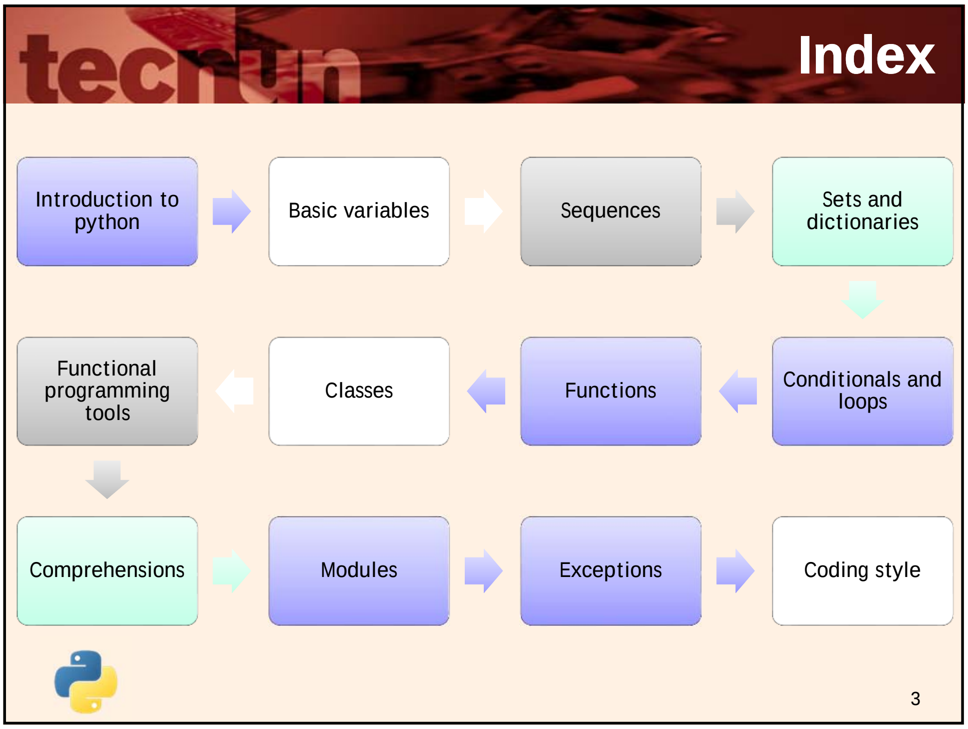
Gazebo (simulatio n)

Navigation

Mapping

Localization

Path planning

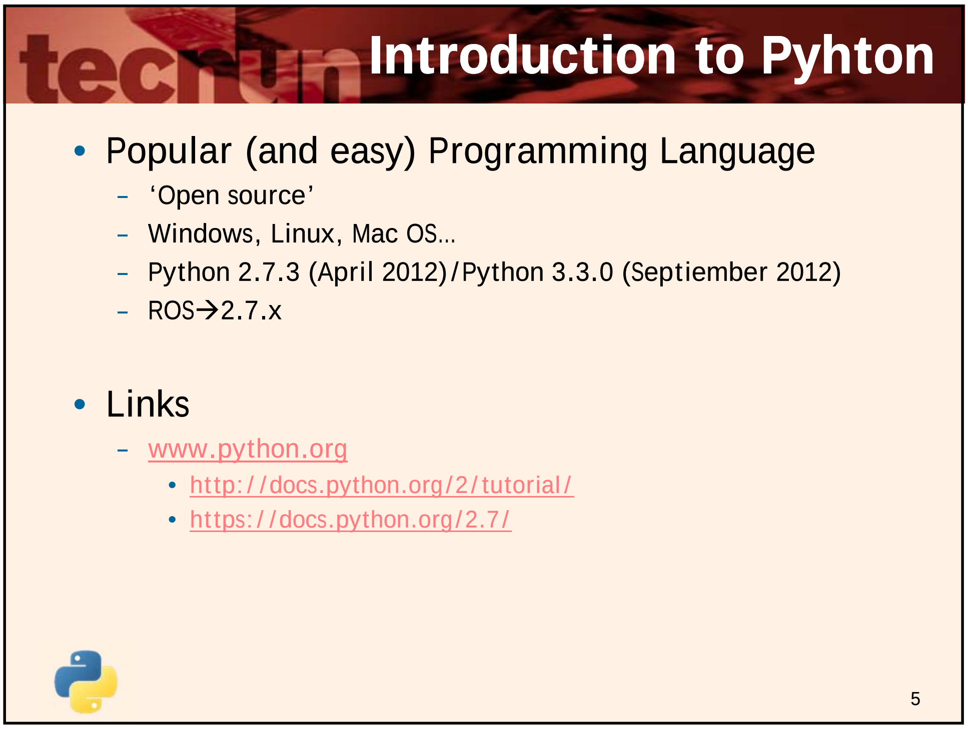


Index

Introduction to python → Basic variables → Sequences → Sets and dictionaries → Conditionals and loops → Functions → Classes → Functional programming tools → Comprehensions → Modules → Exceptions → Coding style

3

Introduction to Python

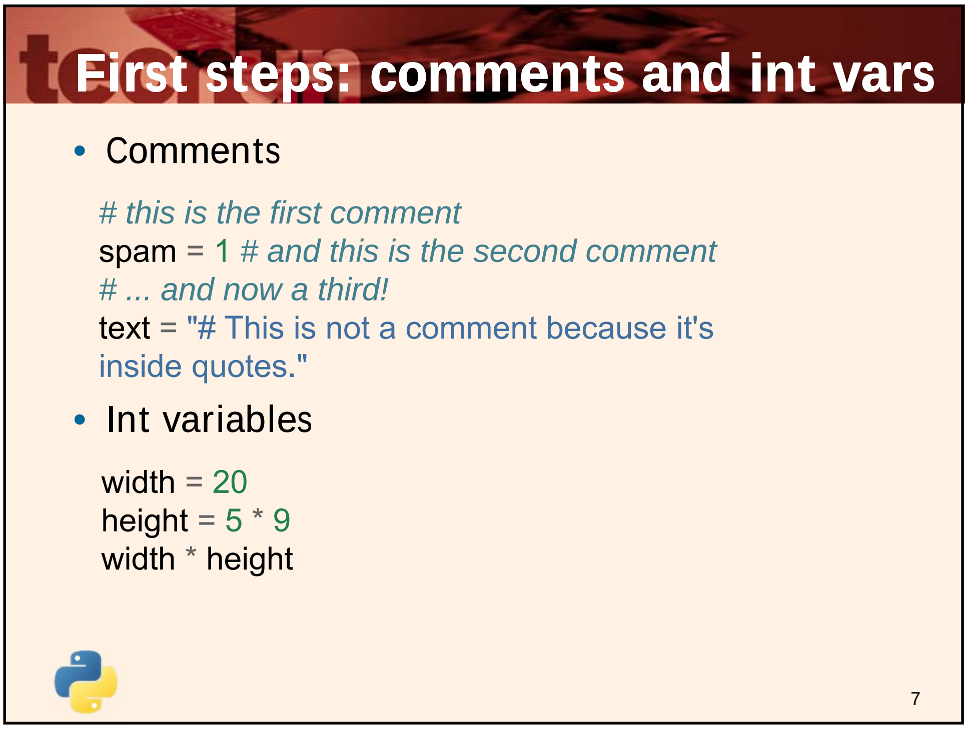


Introduction to Pyhton

- Popular (and easy) Programming Language
  - ‘Open source’
  - Windows, Linux, Mac OS…
  - Python 2.7.3 (April 2012)/Python 3.3.0 (Septiember 2012)
  - ROS→2.7.x
- Links
  - www.python.org
    - http://docs.python.org/2/tutorial/
    - https://docs.python.org/2.7/

5

Variables in Python



First steps: comments and int vars

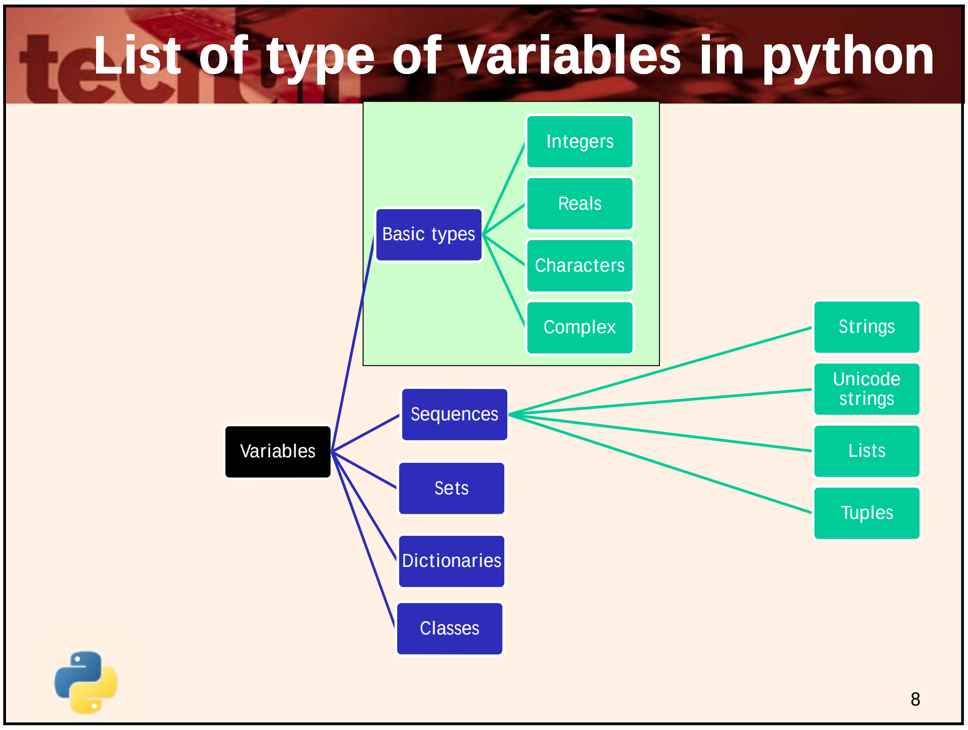
- Comments

```
# this is the first comment
spam = 1 # and this is the second comment
# ... and now a third!
text = "# This is not a comment because it's
inside quotes."
```

- Int variables

```
width = 20
height = 5 * 9
width * height
```

7



List of type of variables in python

Variables

Basic types

Integers

Reals

Characters

Complex

Sequences

Strings

Unicode strings

Lists

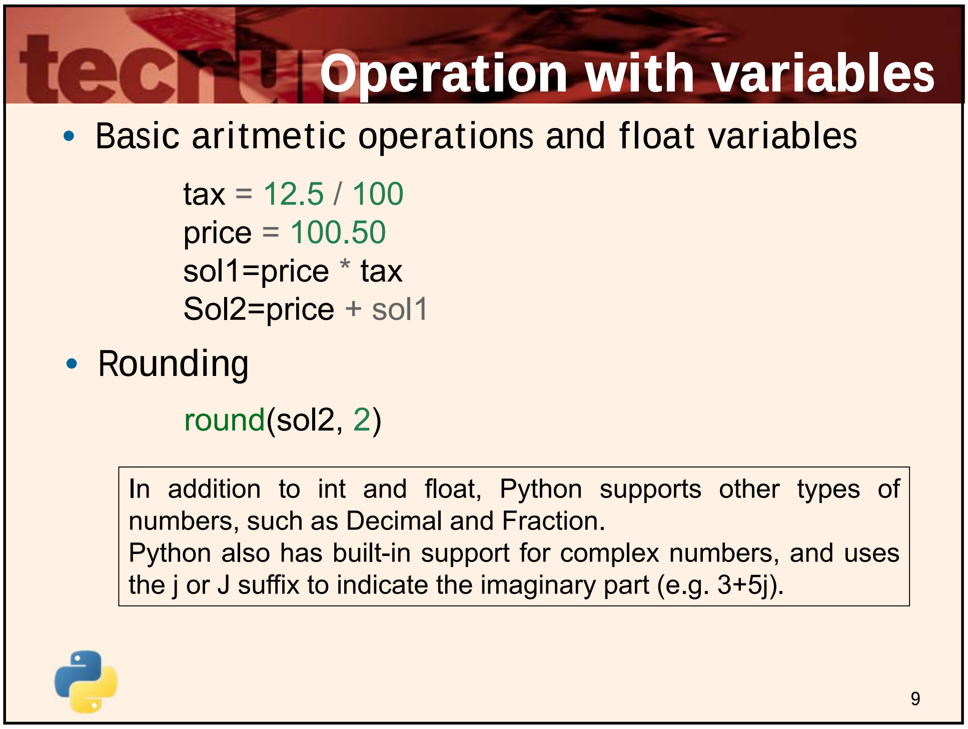
Tuples

Sets

Dictionaries

Classes

8



Operation with variables

- Basic aritmetic operations and float variables

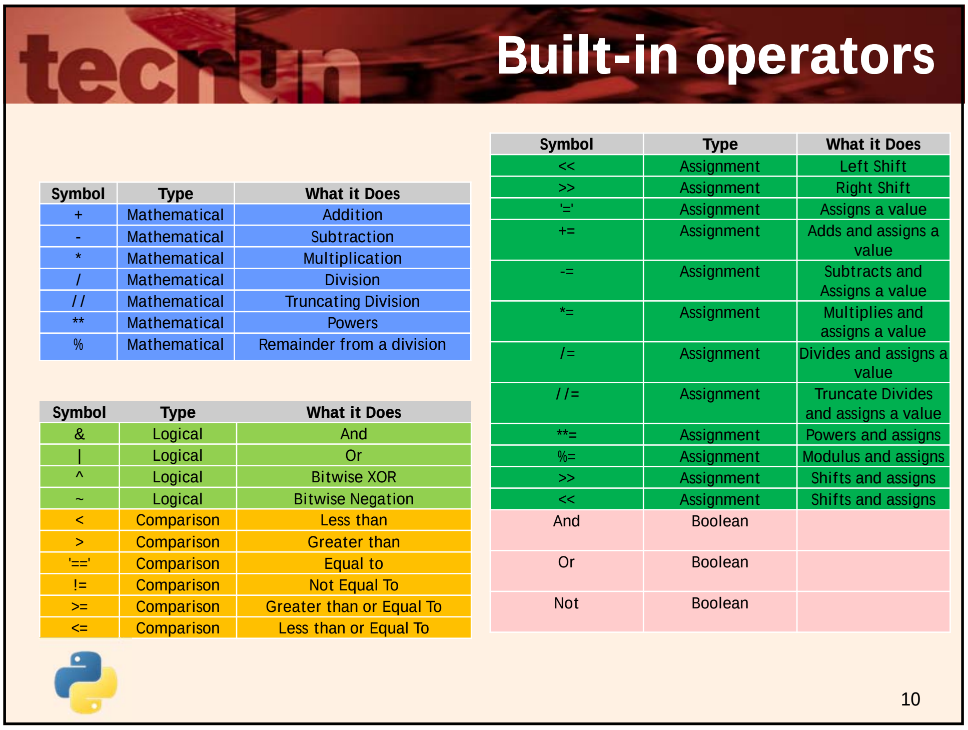
```
tax = 12.5 / 100
price = 100.50
sol1=price * tax
Sol2=price + sol1
```

- Rounding

```
round(sol2, 2)
```

In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction.
Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

9



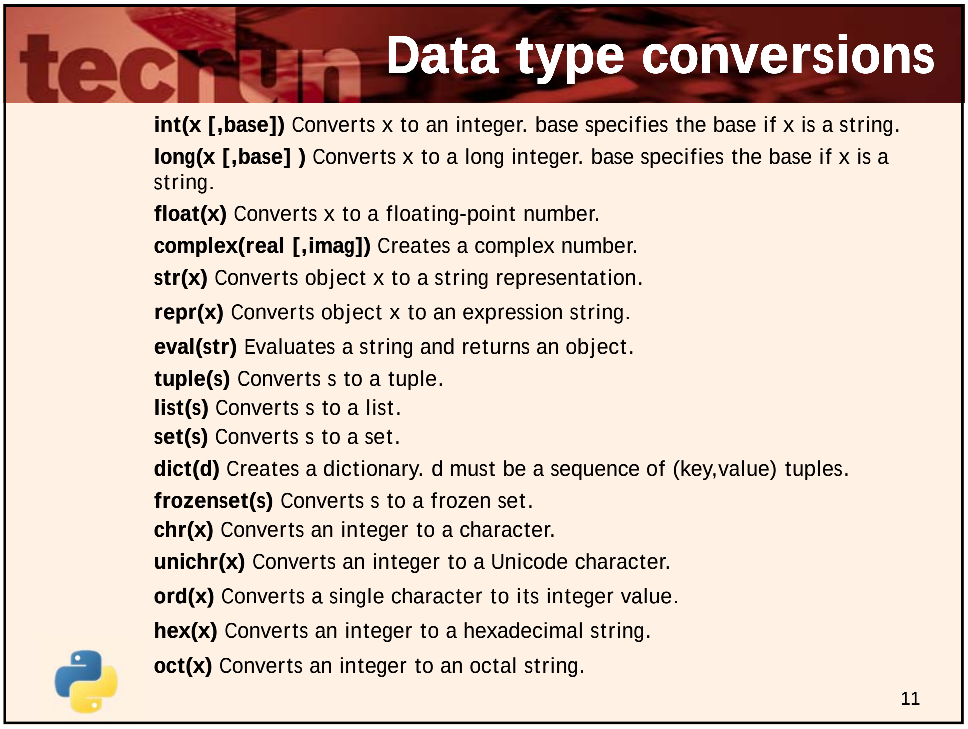
Built-in operators

Symbol	Type	What it Does
+	Mathematical	Addition
-	Mathematical	Subtraction
*	Mathematical	Multiplication
/	Mathematical	Division
/ /	Mathematical	Truncating Division
**	Mathematical	Powers
%	Mathematical	Remainder from a division

Symbol	Type	What it Does
&	Logical	And
|	Logical	Or
^	Logical	Bitwise XOR
~	Logical	Bitwise Negation
<	Comparison	Less than
>	Comparison	Greater than
'=='	Comparison	Equal to
!=	Comparison	Not Equal To
>=	Comparison	Greater than or Equal To
<=	Comparison	Less than or Equal To

Symbol	Type	What it Does
<<	Assignment	Left Shift
>>	Assignment	Right Shift
'='	Assignment	Assigns a value
+=	Assignment	Adds and assigns a value
-=	Assignment	Subtracts and Assigns a value
*=	Assignment	Multiplies and assigns a value
/=	Assignment	Divides and assigns a value
/ /=	Assignment	Truncate Divides and assigns a value
**=	Assignment	Powers and assigns
%=	Assignment	Modulus and assigns
>>	Assignment	Shifts and assigns
<<	Assignment	Shifts and assigns
And	Boolean	
Or	Boolean	
Not	Boolean	

10



Data type conversions

int(x [,base]) Converts x to an integer. base specifies the base if x is a string.

long(x [,base]) Converts x to a long integer. base specifies the base if x is a string.

float(x) Converts x to a floating-point number.

complex(real [,imag]) Creates a complex number.

str(x) Converts object x to a string representation.

repr(x) Converts object x to an expression string.

eval(str) Evaluates a string and returns an object.

tuple(s) Converts s to a tuple.

list(s) Converts s to a list.

set(s) Converts s to a set.

dict(d) Creates a dictionary. d must be a sequence of (key,value) tuples.

frozenset(s) Converts s to a frozen set.

chr(x) Converts an integer to a character.

unichr(x) Converts an integer to a Unicode character.

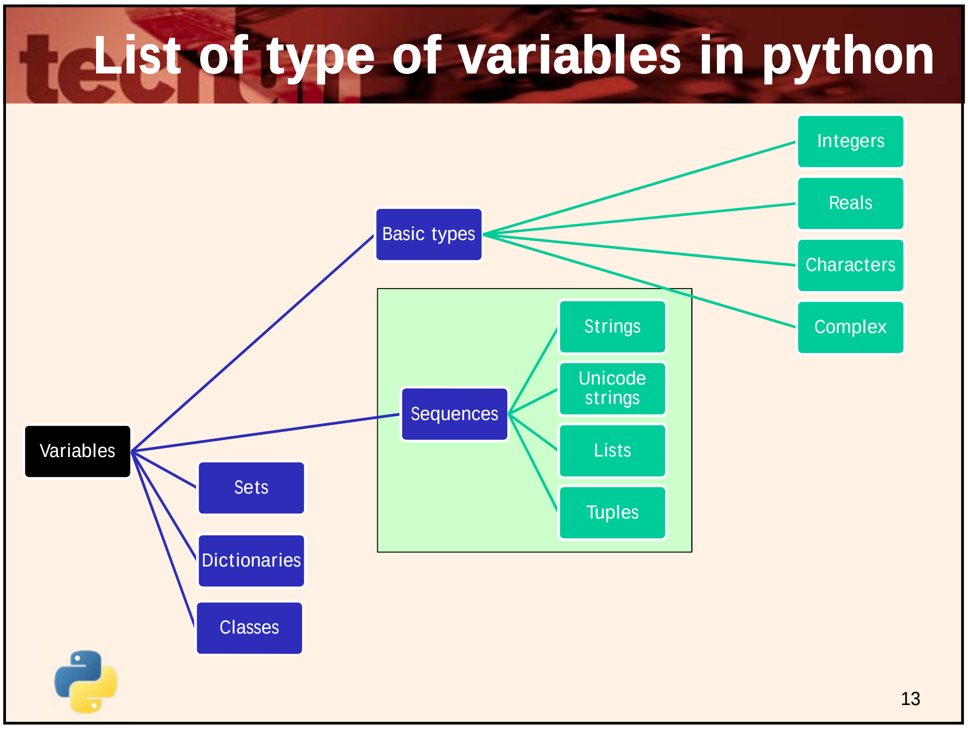
ord(x) Converts a single character to its integer value.

hex(x) Converts an integer to a hexadecimal string.

oct(x) Converts an integer to an octal string.

11

Sequences in Python



List of type of variables in python

Variables

Basic types

Integers

Reals

Characters

Complex

Sequences

Strings

Unicode strings

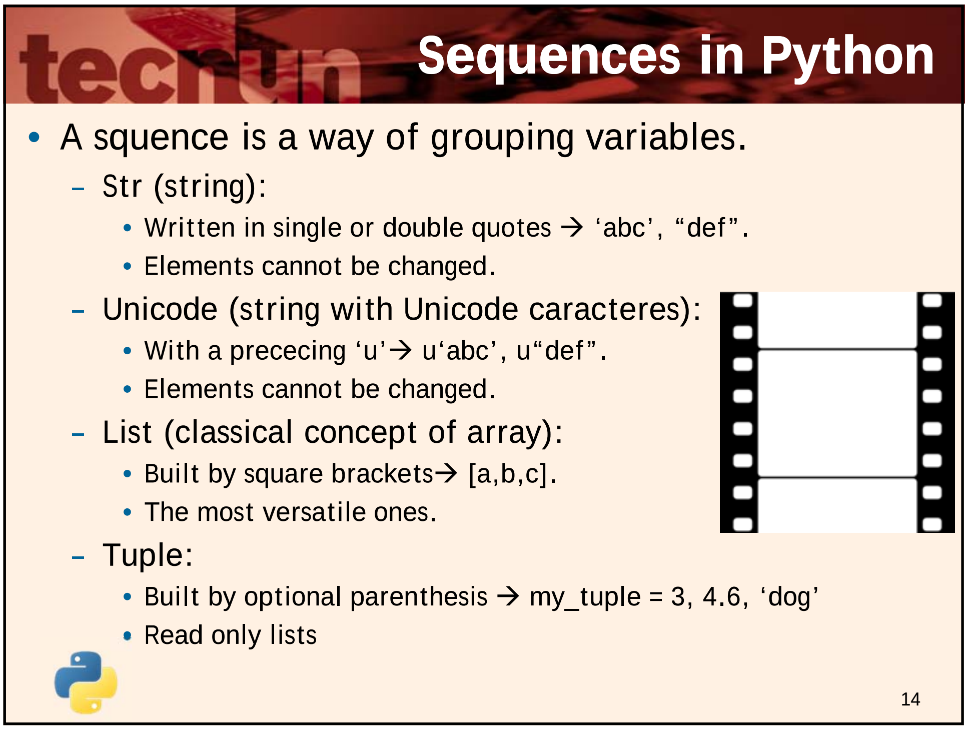
Lists

Tuples

Sets

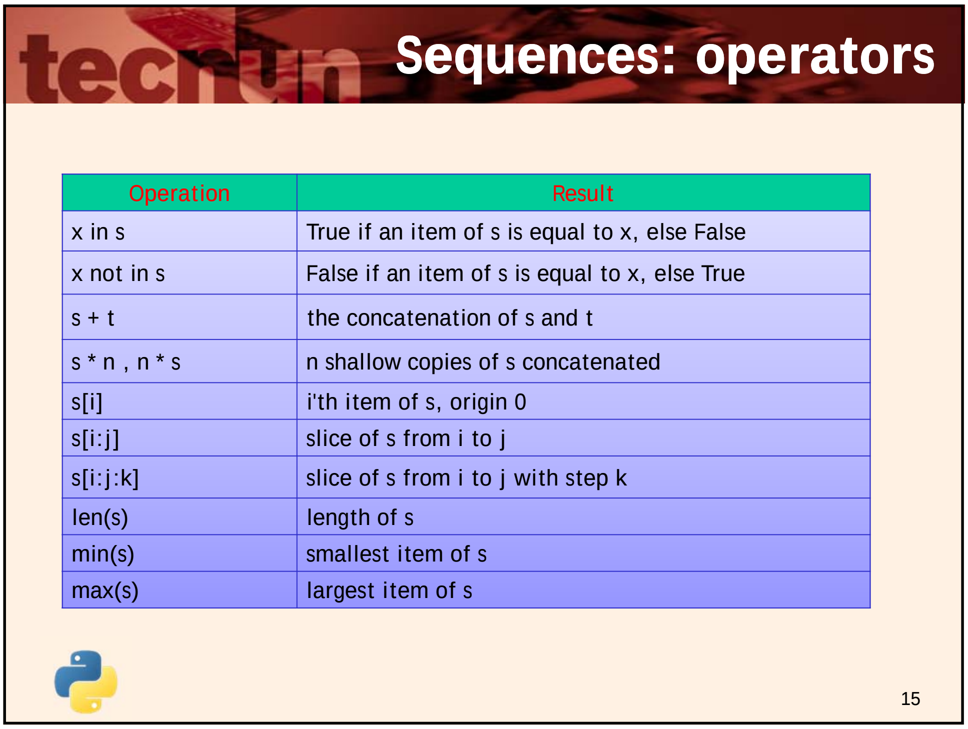
Dictionaries

Classes



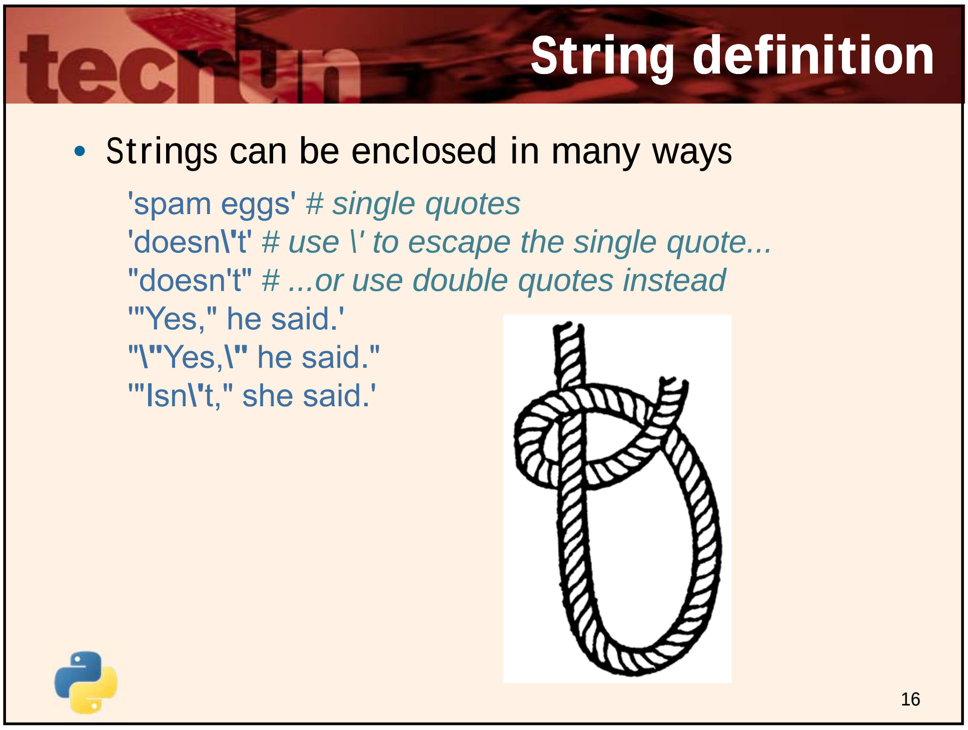
Sequences in Python

- A squence is a way of grouping variables.
  - Str (string):
    - Written in single or double quotes → ‘abc’, “def”.
    - Elements cannot be changed.
  - Unicode (string with Unicode caracteres):
    - With a prececing ‘u’→ u‘abc’, u“def”.
    - Elements cannot be changed.
  - List (classical concept of array):
    - Built by square brackets→ [a,b,c].
    - The most versatile ones.
  - Tuple:
    - Built by optional parenthesis → my_tuple = 3, 4.6, ‘dog’
    - Read only lists



Sequences: operators

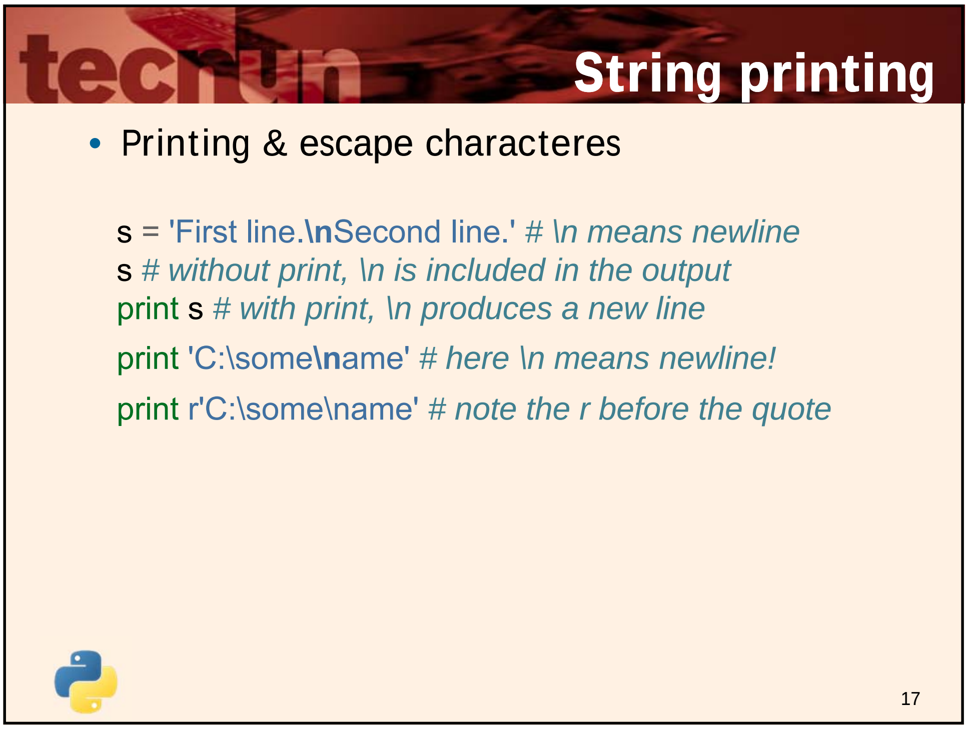
Operation	Result
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s + t	the concatenation of s and t
s * n , n * s	n shallow copies of s concatenated
s[i]	i'th item of s, origin 0
s[i:j]	slice of s from i to j
s[i:j:k]	slice of s from i to j with step k
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s



String definition

- Strings can be enclosed in many ways

```
'spam eggs' # single quotes
'doesn\'t' # use \' to escape the single quote...
"doesn't" # ...or use double quotes instead
'"Yes," he said.'
"\"Yes,\" he said."
'"Isn\'t," she said.'
```

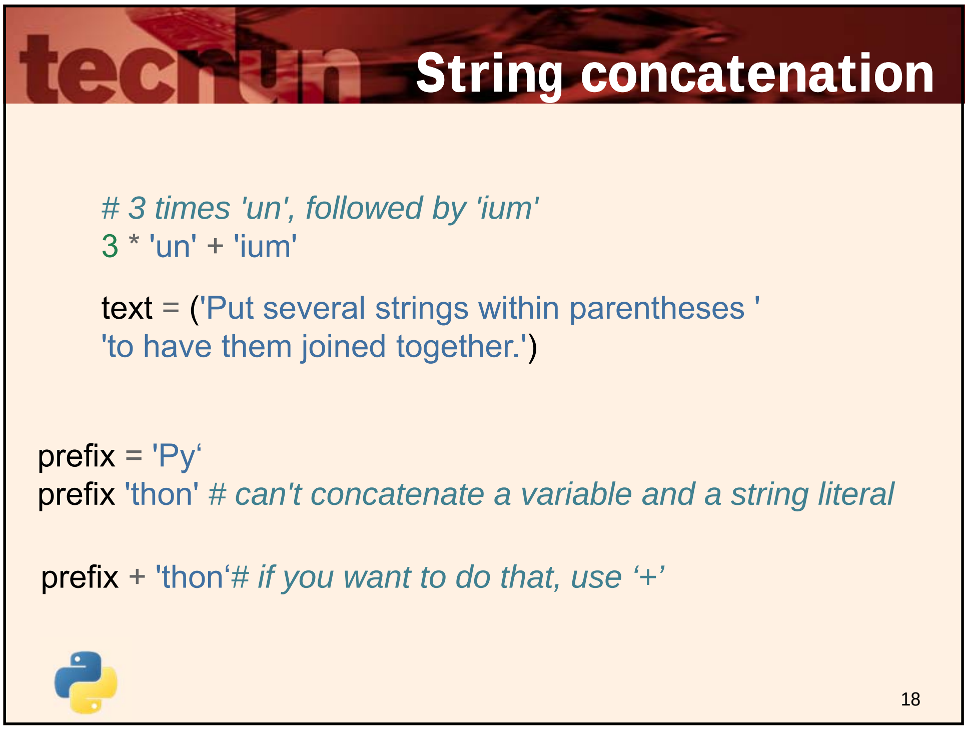


String printing

- Printing & escape characteres

```
s = 'First line.\nSecond line.' # \n means newline
s # without print, \n is included in the output
print s # with print, \n produces a new line
print 'C:\some\name' # here \n means newline!
print r'C:\some\name' # note the r before the quote
```

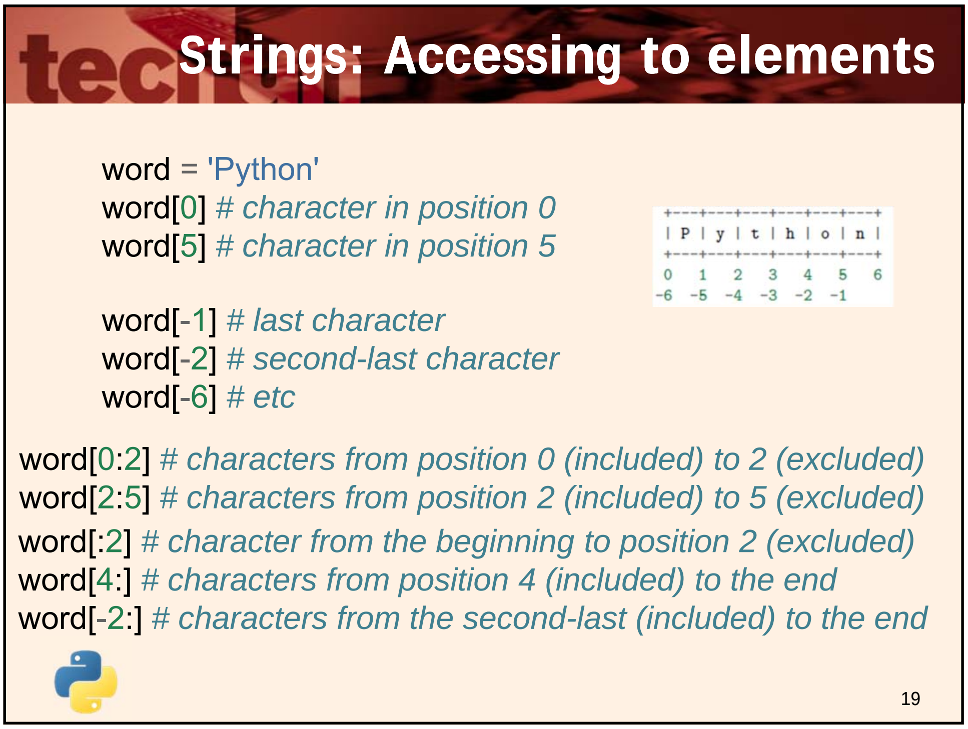
17



String concatenation

```
# 3 times 'un', followed by 'ium'
3 * 'un' + 'ium'

text = ('Put several strings within parentheses '
'to have them joined together.')

prefix = 'Py‘
prefix 'thon' # can't concatenate a variable and a string literal

prefix + 'thon‘# if you want to do that, use ‘+’
```

18



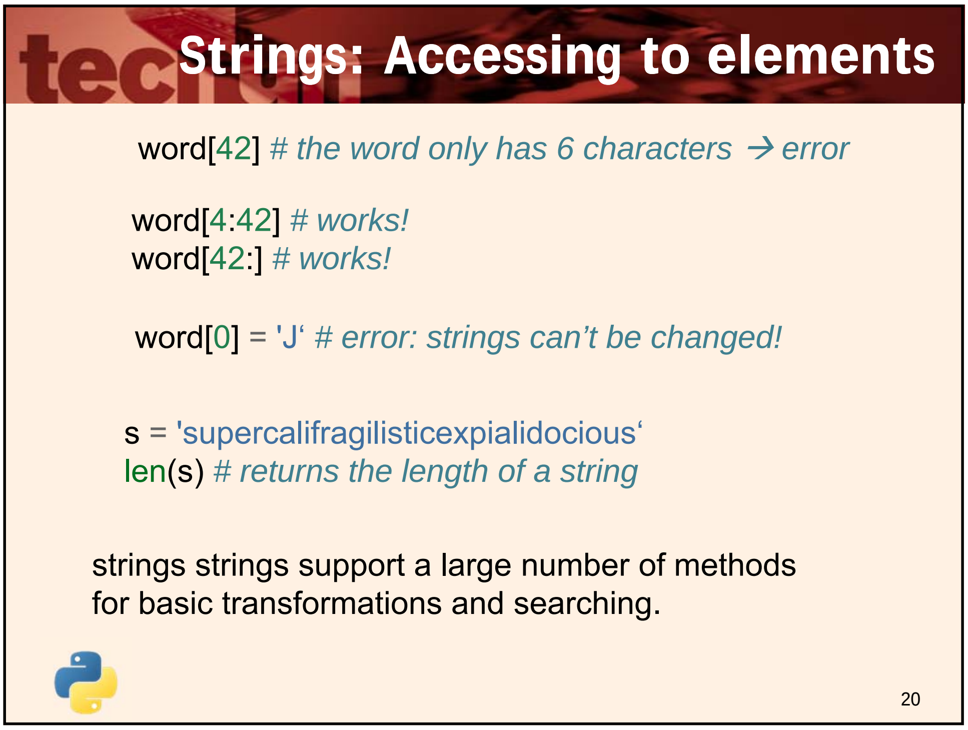
Strings: Accessing to elements

```
word = 'Python'
word[0] # character in position 0
word[5] # character in position 5

word[-1] # last character
word[-2] # second-last character
word[-6] # etc
```

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

```
word[0:2] # characters from position 0 (included) to 2 (excluded)
word[2:5] # characters from position 2 (included) to 5 (excluded)
word[:2] # character from the beginning to position 2 (excluded)
word[4:] # characters from position 4 (included) to the end
word[-2:] # characters from the second-last (included) to the end
```

19

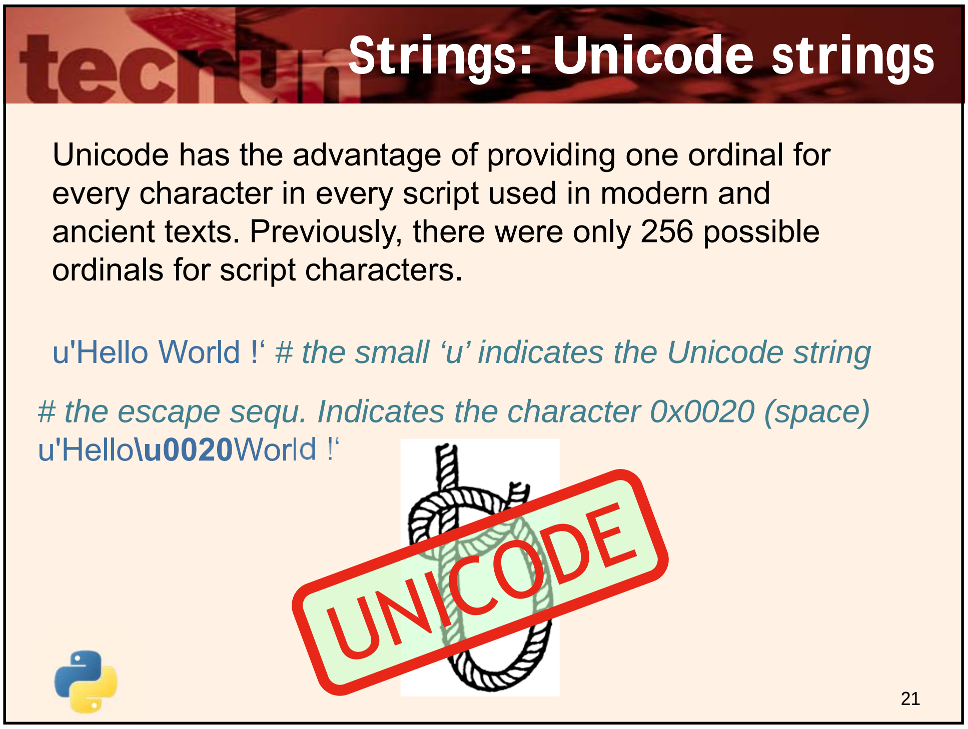


Strings: Accessing to elements

```
word[42] # the word only has 6 characters → error

word[4:42] # works!
word[42:] # works!

word[0] = 'J‘ # error: strings can’t be changed!

s = 'supercalifragilisticexpialidocious‘
len(s) # returns the length of a string
```

strings strings support a large number of methods
for basic transformations and searching.

20

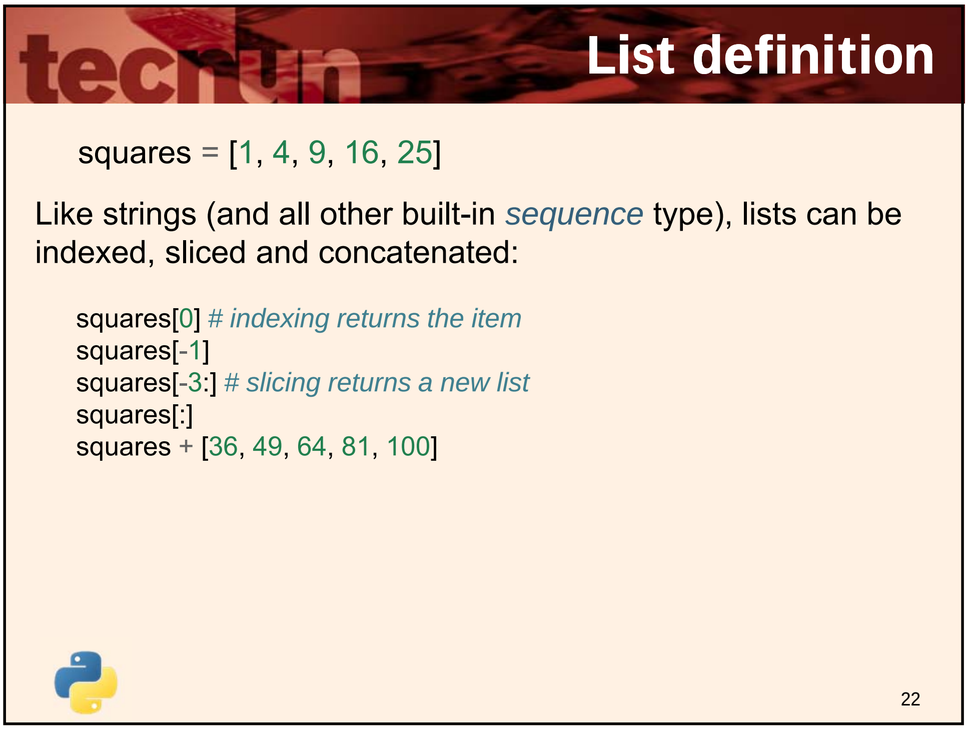


Strings: Unicode strings

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters.

```
u'Hello World !‘ # the small ‘u’ indicates the Unicode string

# the escape sequ. Indicates the character 0x0020 (space)
u'Hello\u0020World !‘
```

21



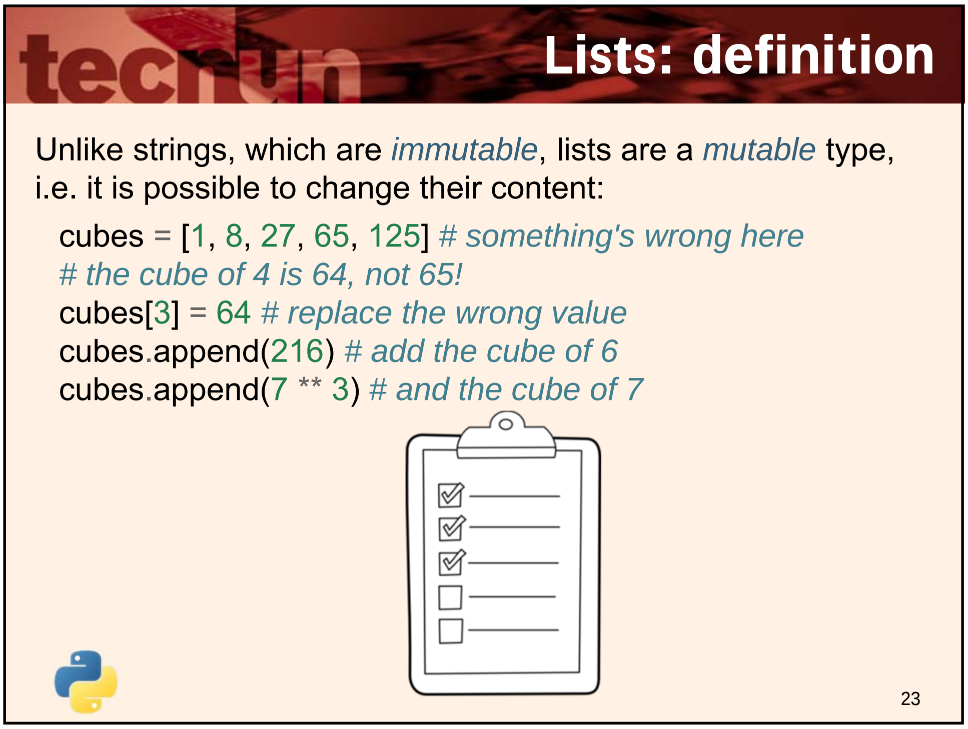
List definition

```
squares = [1, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* type), lists can be indexed, sliced and concatenated:

```
squares[0] # indexing returns the item
squares[-1]
squares[-3:] # slicing returns a new list
squares[:]
squares + [36, 49, 64, 81, 100]
```

22

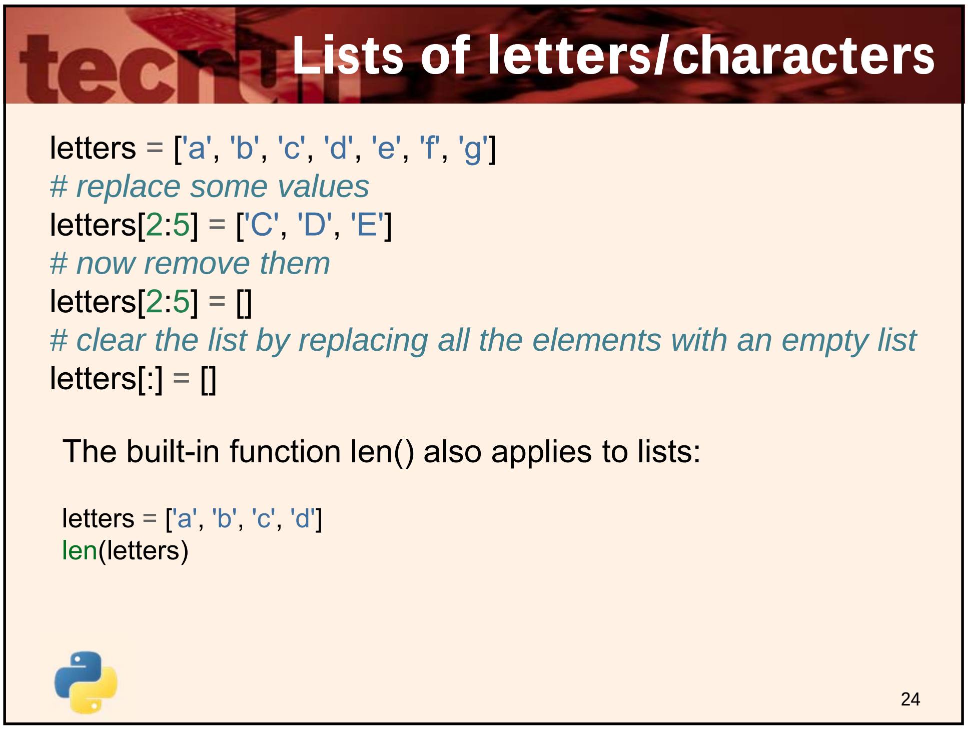


Lists: definition

Unlike strings, which are *immutable*, lists are a *mutable* type, i.e. it is possible to change their content:

```
cubes = [1, 8, 27, 65, 125] # something's wrong here
# the cube of 4 is 64, not 65!
cubes[3] = 64 # replace the wrong value
cubes.append(216) # add the cube of 6
cubes.append(7 ** 3) # and the cube of 7
```

23



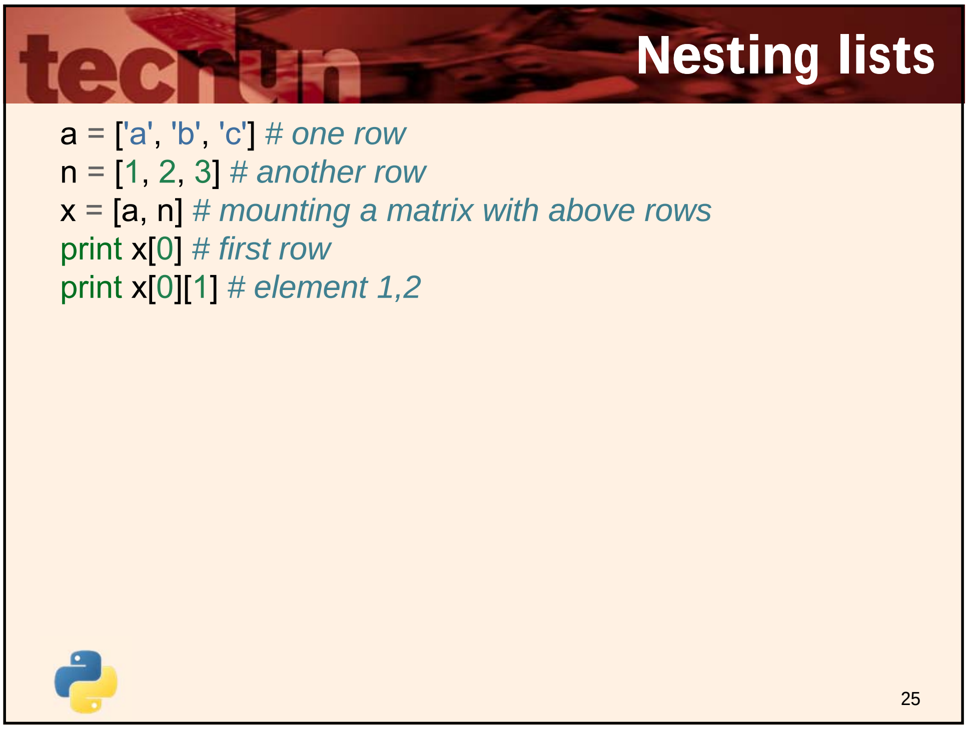
Lists of letters/characters

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
# replace some values
letters[2:5] = ['C', 'D', 'E']
# now remove them
letters[2:5] = []
# clear the list by replacing all the elements with an empty list
letters[:] = []
```

The built-in function len() also applies to lists:

```
letters = ['a', 'b', 'c', 'd']
len(letters)
```

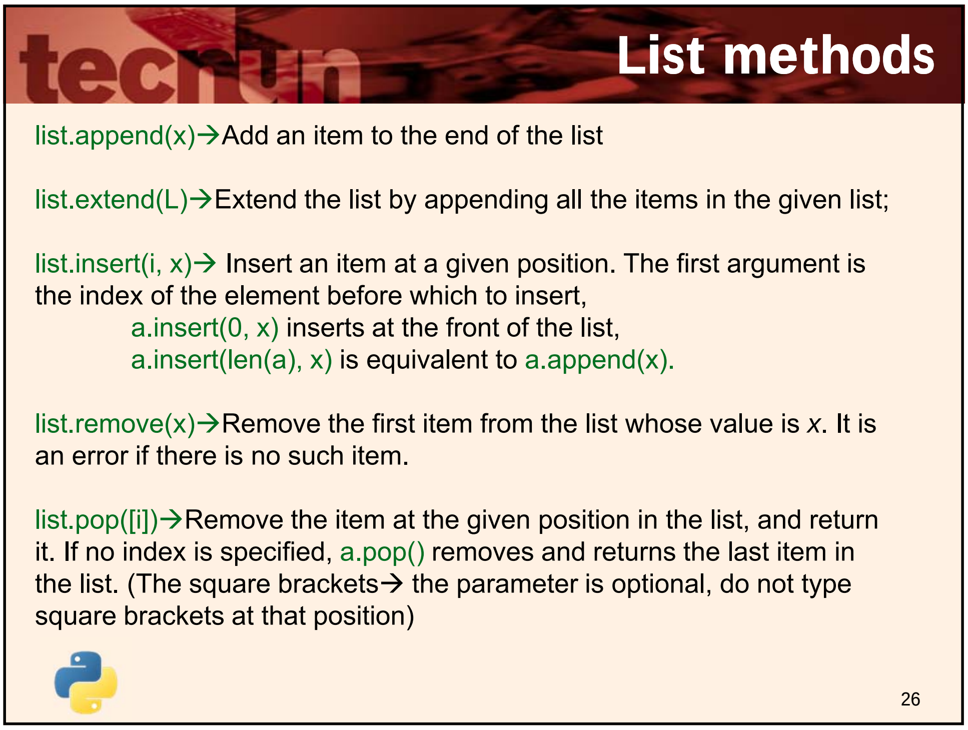
24



Nesting lists

```
a = ['a', 'b', 'c'] # one row
n = [1, 2, 3] # another row
x = [a, n] # mounting a matrix with above rows
print x[0] # first row
print x[0][1] # element 1,2
```

25



List methods

list.append(x)→Add an item to the end of the list

list.extend(L)→Extend the list by appending all the items in the given list;

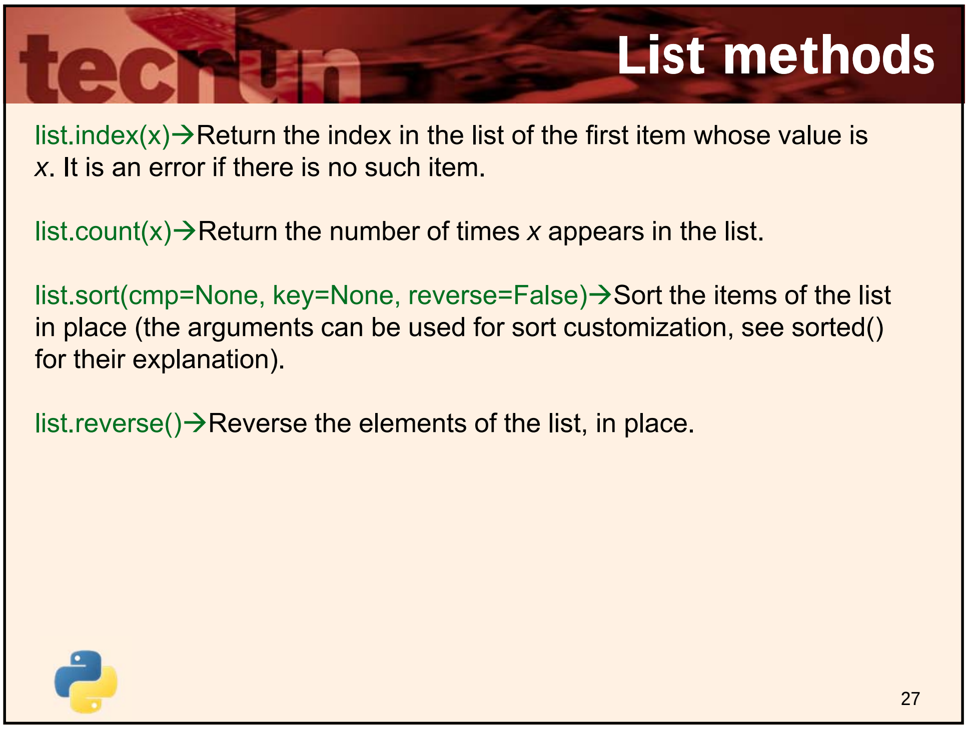
list.insert(i, x)→ Insert an item at a given position. The first argument is the index of the element before which to insert,

a.insert(0, x) inserts at the front of the list,
a.insert(len(a), x) is equivalent to a.append(x).

list.remove(x)→Remove the first item from the list whose value is *x*. It is an error if there is no such item.

list.pop([i])→Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets→ the parameter is optional, do not type square brackets at that position)

26



List methods

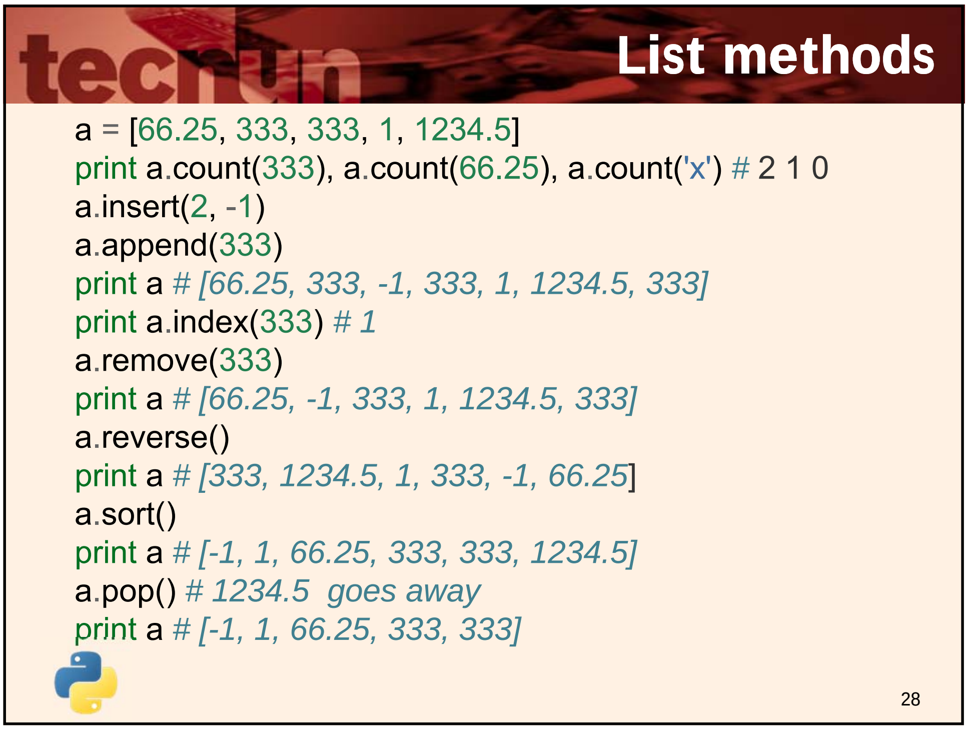
list.index(x)→Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

list.count(x)→Return the number of times *x* appears in the list.

list.sort(cmp=None, key=None, reverse=False)→Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

list.reverse()→Reverse the elements of the list, in place.

27



List methods

```
a = [66.25, 333, 333, 1, 1234.5]
print a.count(333), a.count(66.25), a.count('x') # 2 1 0
a.insert(2, -1)
a.append(333)
print a # [66.25, 333, -1, 333, 1, 1234.5, 333]
print a.index(333) # 1
a.remove(333)
print a # [66.25, -1, 333, 1, 1234.5, 333]
a.reverse()
print a # [333, 1234.5, 1, 333, -1, 66.25]
a.sort()
print a # [-1, 1, 66.25, 333, 333, 1234.5]
a.pop() # 1234.5  goes away
print a # [-1, 1, 66.25, 333, 333]
```

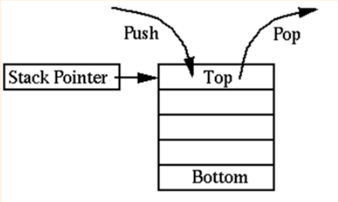
28

List: deleting an element

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print a # [1, 66.25, 333, 333, 1234.5]
del a[2:4]
print a # [1, 66.25, 1234.5]
del a[:]
print  a # [ ]

del a # deletes all variables in the list
```

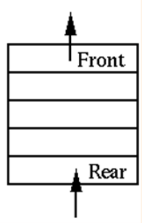
Lists: stack (LIFO)

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack # [3, 4, 5, 6, 7]
stack.pop() #7
stack # [3, 4, 5, 6]
stack.pop() # 6
stack.pop() # 5
stack #[3, 4]
```



Lists: queues (FIFO)

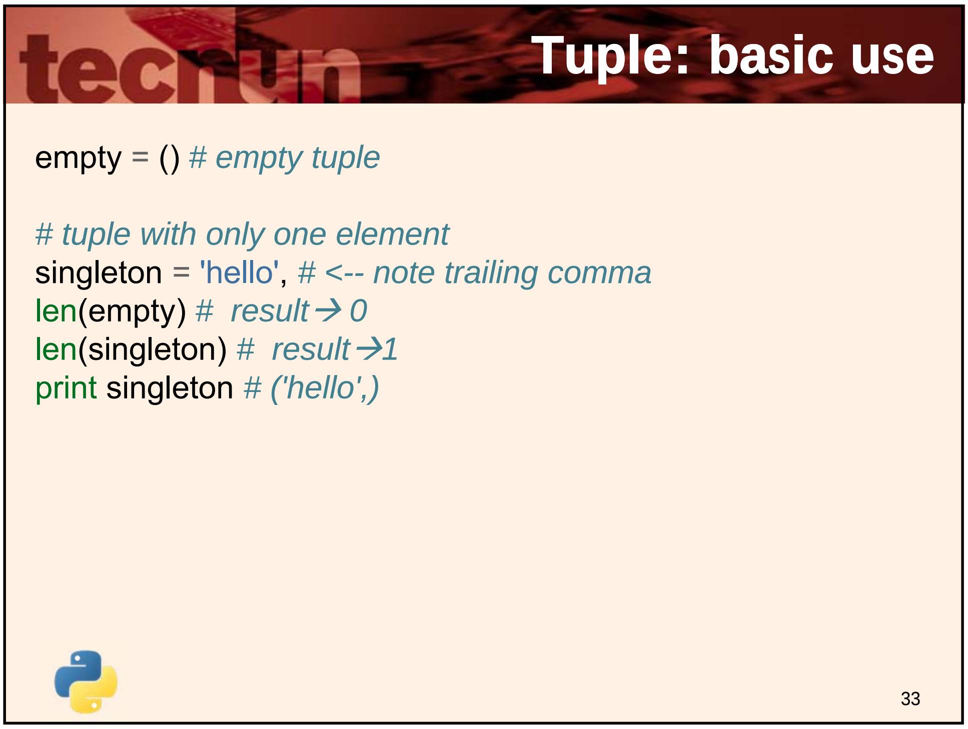
```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
queue.popleft() # The first to arrive now leaves → 'Eric'
queue.popleft() # The second to arrive now leaves → 'John'
queue # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```



Tuple: basic use

$(a_1,a_2,a_3, \ldots a_n)$

```
t = 12345, 54321, 'hello!'
print t[0] # 12345
print  t # (12345, 54321, 'hello!')
>>> # Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
print u # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
 # Tuples are immutable:
t[0] = 88888  # ERROR
# but they can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
print v # ([1, 2, 3], [3, 2, 1])
x, y, z = t # unpack the tuple
```

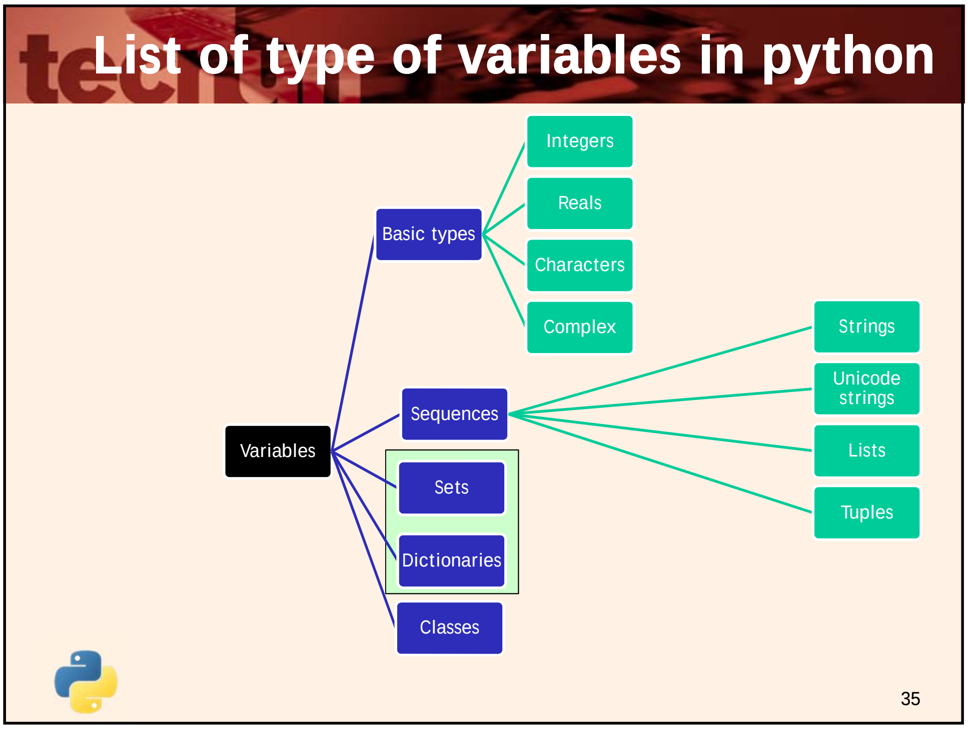


Tuple: basic use

```
empty = () # empty tuple

# tuple with only one element
singleton = 'hello', # <-- note trailing comma
len(empty) #  result→ 0
len(singleton) #  result→1
print singleton # ('hello',)
```

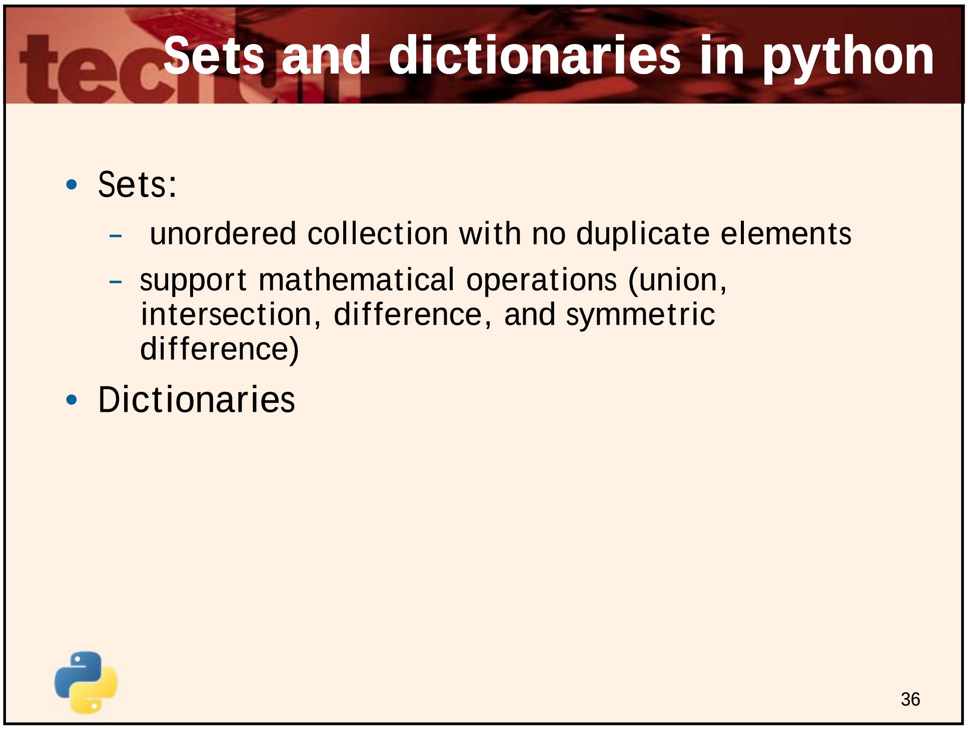
33

SET and Dictionaries in Python



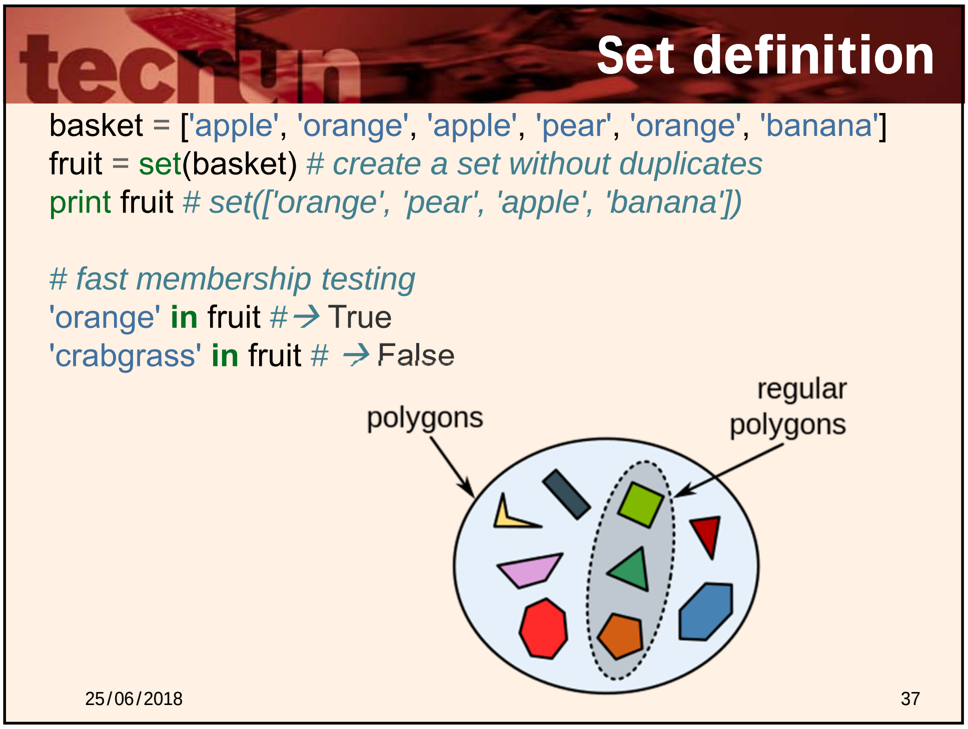
List of type of variables in python

- Variables
  - Basic types
    - Integers
    - Reals
    - Characters
    - Complex
  - Sequences
    - Strings
    - Unicode strings
    - Lists
    - Tuples
  - Sets
  - Dictionaries
  - Classes



Sets and dictionaries in python

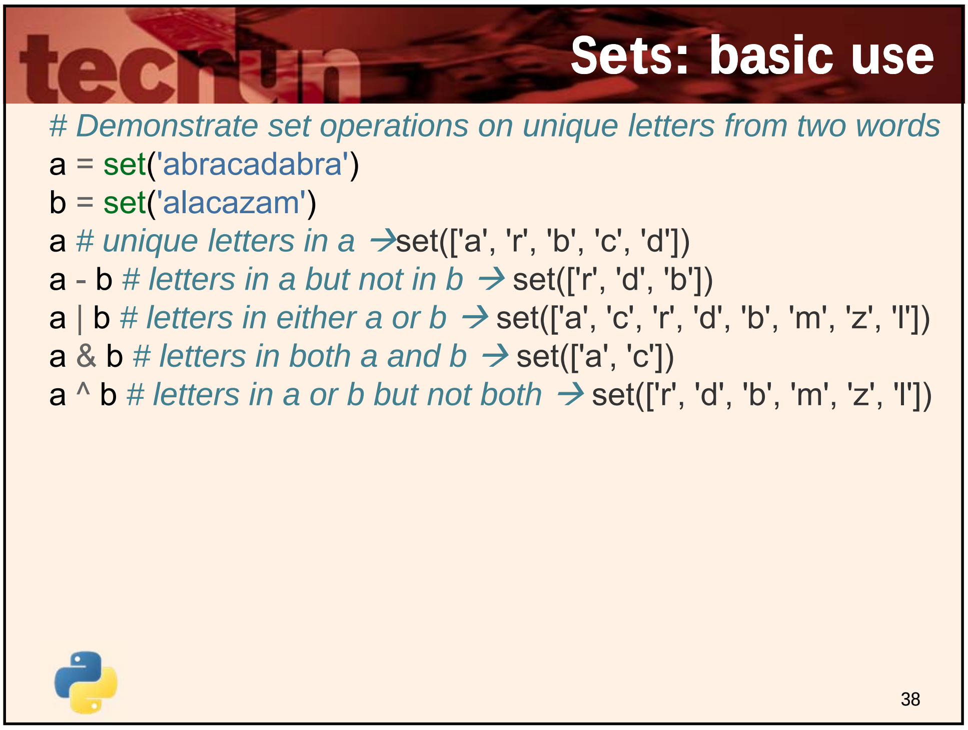
- Sets:
  - unordered collection with no duplicate elements
  - support mathematical operations (union, intersection, difference, and symmetric difference)
- Dictionaries



Set definition

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
fruit = set(basket) # create a set without duplicates
print fruit # set(['orange', 'pear', 'apple', 'banana'])

# fast membership testing
'orange' in fruit #→ True
'crabgrass' in fruit # → False
```

25/06/2018 37



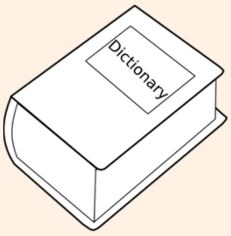
Sets: basic use

```
# Demonstrate set operations on unique letters from two words
a = set('abracadabra')
b = set('alacazam')
a # unique letters in a →set(['a', 'r', 'b', 'c', 'd'])
a - b # letters in a but not in b → set(['r', 'd', 'b'])
a | b # letters in either a or b → set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
a & b # letters in both a and b → set(['a', 'c'])
a ^ b # letters in a or b but not both → set(['r', 'd', 'b', 'm', 'z', 'l'])
```

38

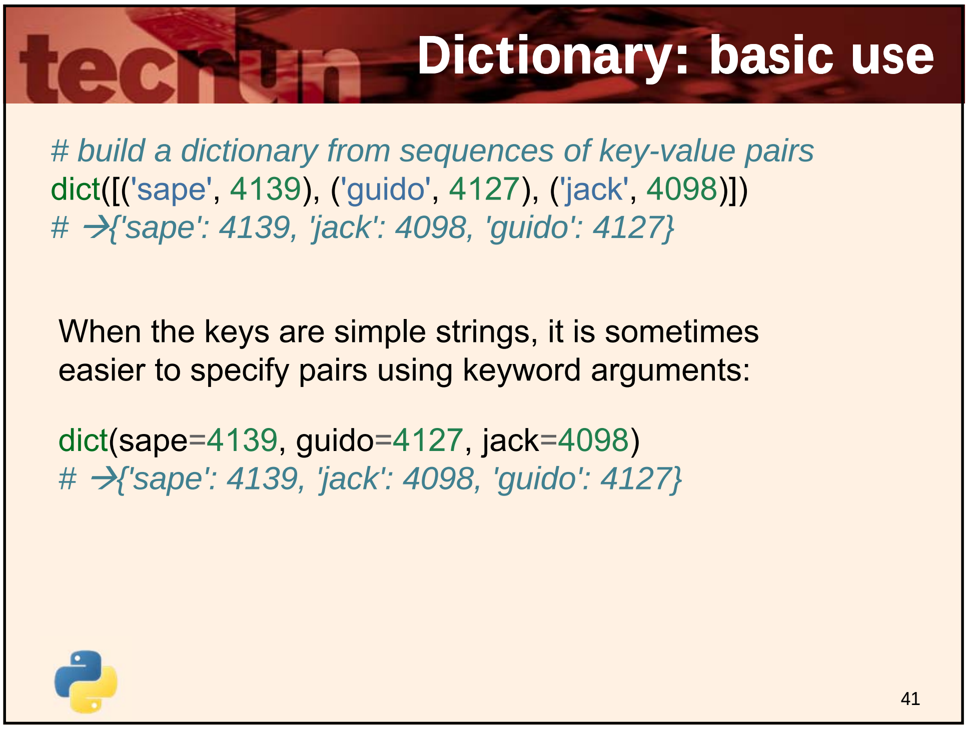
Dictionary

- Associative arrays/associative memories
- Consists of a kind of set of *key:value* pairs
- Indexed by the key, not by a number
- A dictionary key can be almost any Python type, but are usually numbers or strings.
- Values can be any arbitrary Python object.
- Values assigned and accessed by square braces ([])
- Enclosed by curly braces ({ })



Dictionary: basic use

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127 # add a new entry
print tel # {'sape': 4139, 'guido': 4127, 'jack': 4098}
print tel['jack'] # 4098
del tel['sape'] # delete the element
tel['irv'] = 4127
print tel # {'guido': 4127, 'irv': 4127, 'jack': 4098}
tel.keys() # print the keys → ['guido', 'irv', 'jack']
'guido' in tel # find outs if a key exists→True
```



Dictionary: basic use

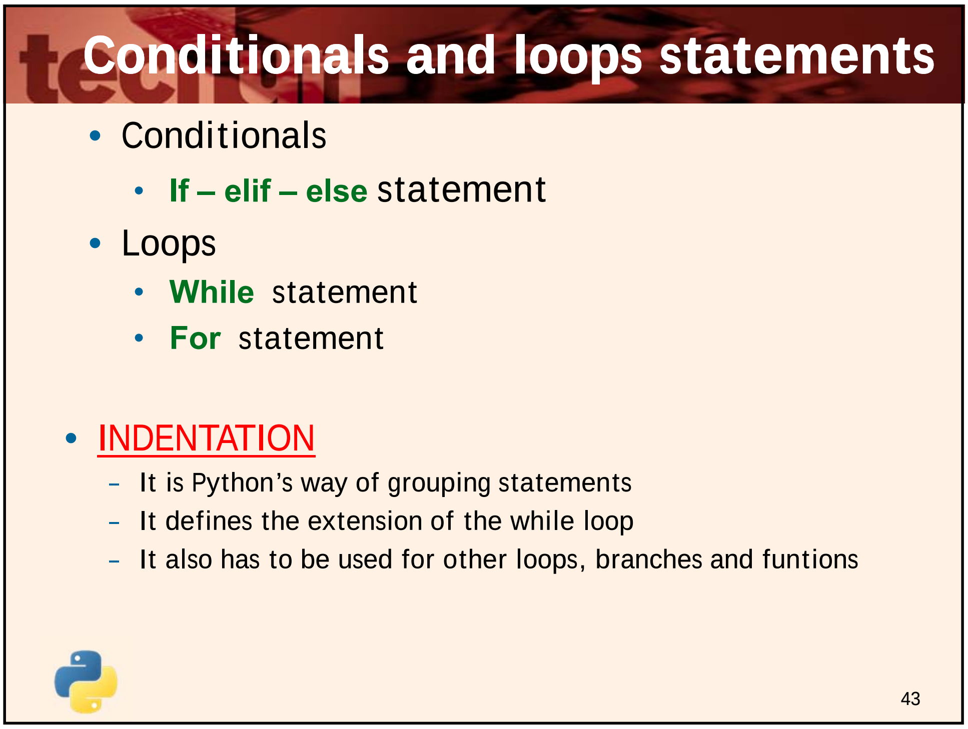
```
# build a dictionary from sequences of key-value pairs
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
# →{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098)
# →{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

41

Conditionals and loops in Python

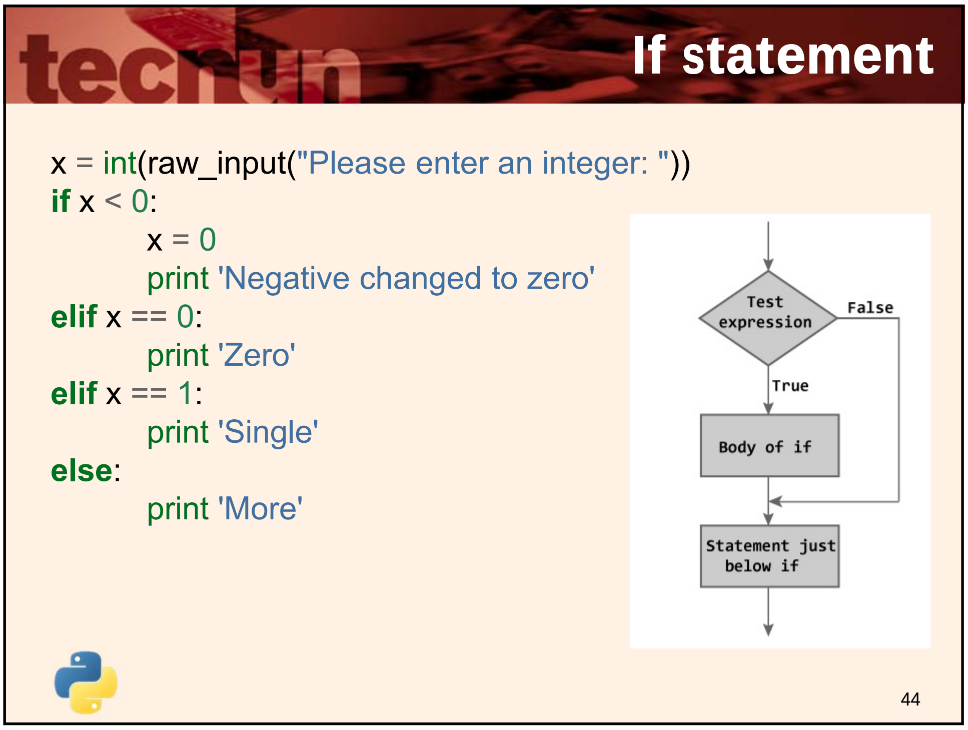


Conditionals and loops statements

- Conditionals
  - **If – elif – else** statement
- Loops
  - **While** statement
  - **For** statement

- INDENTATION
  - It is Python's way of grouping statements
  - It defines the extension of the while loop
  - It also has to be used for other loops, branches and funtions

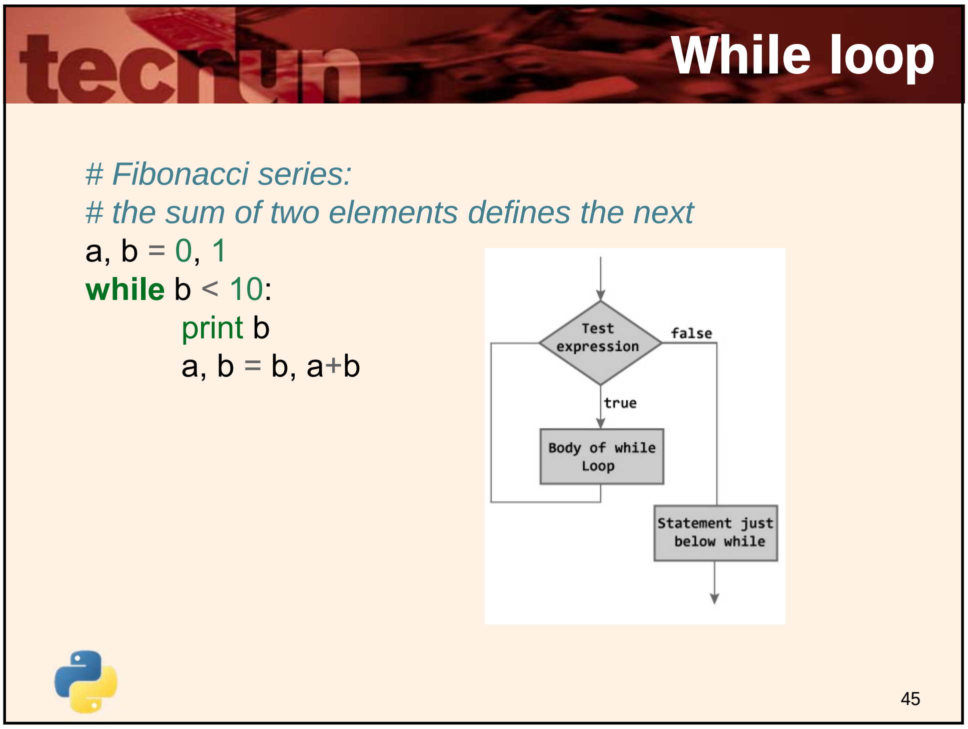
43



If statement

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
        x = 0
        print 'Negative changed to zero'
elif x == 0:
        print 'Zero'
elif x == 1:
        print 'Single'
else:
        print 'More'
```

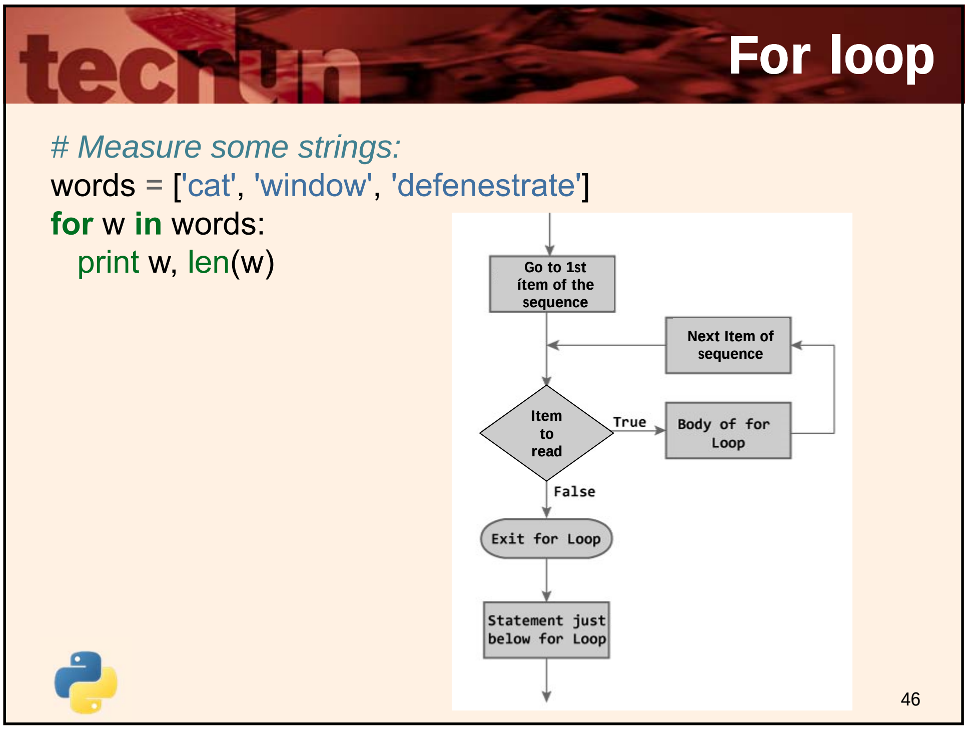
44



While loop

```
# Fibonacci series:
# the sum of two elements defines the next
a, b = 0, 1
while b < 10:
        print b
        a, b = b, a+b
```

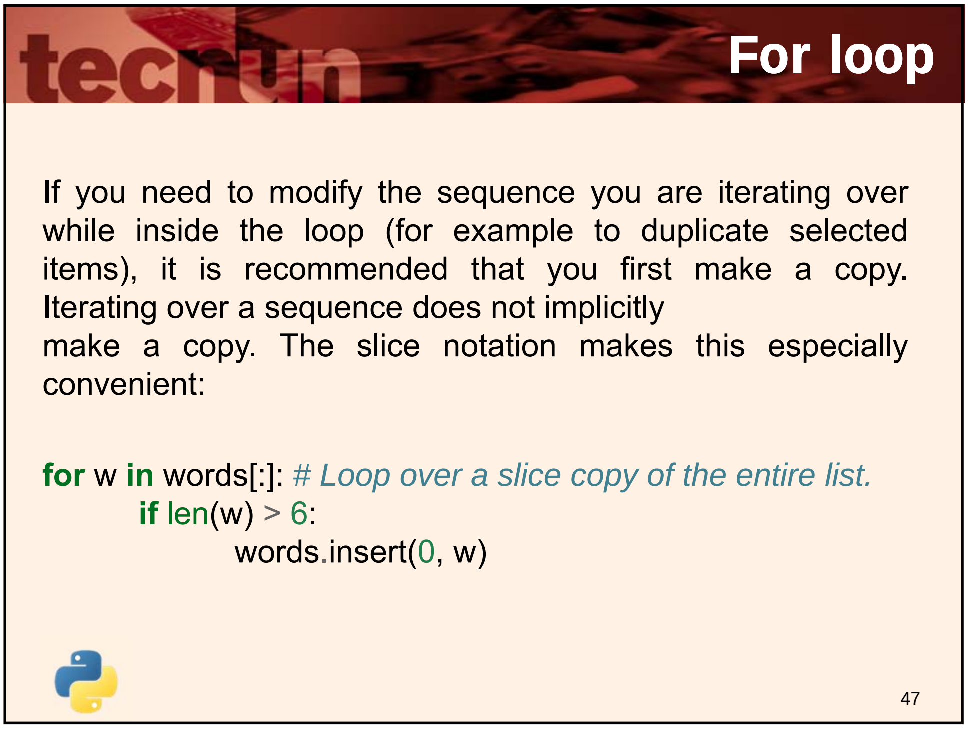
45



For loop

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
   print w, len(w)
```

46

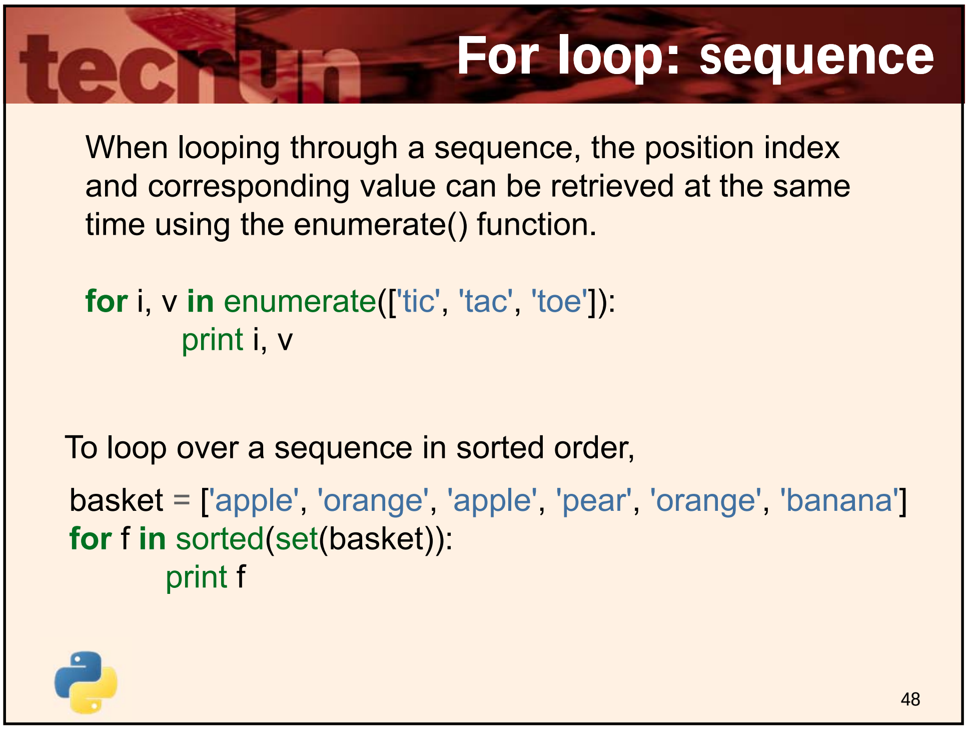


For loop

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
for w in words[:]: # Loop over a slice copy of the entire list.
        if len(w) > 6:
                words.insert(0, w)
```

47



For loop: sequence

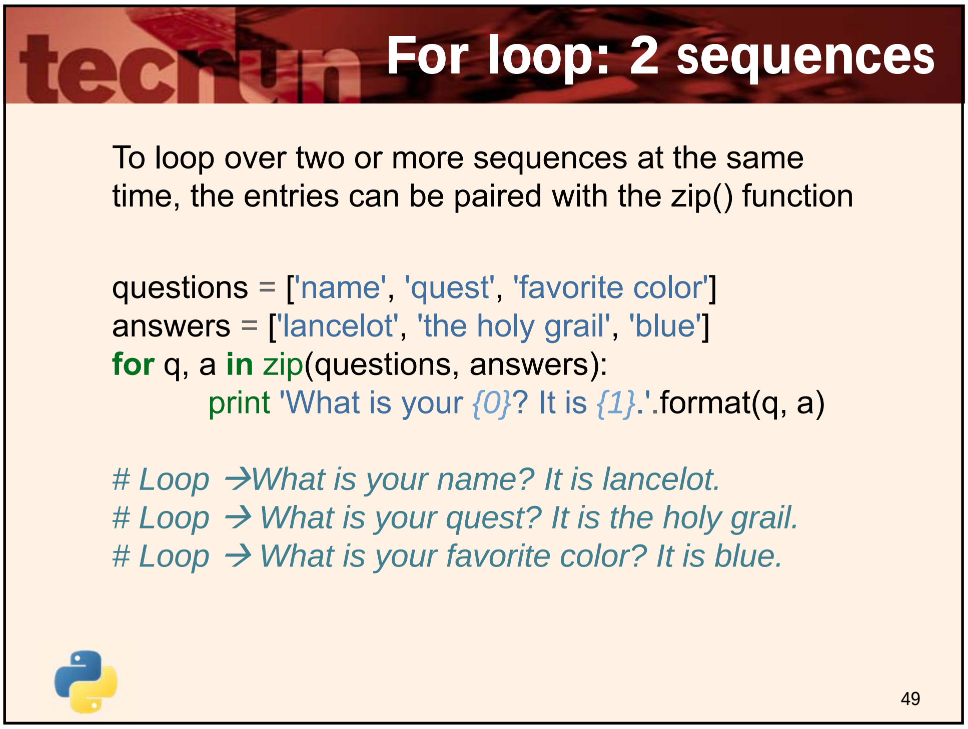
When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
for i, v in enumerate(['tic', 'tac', 'toe']):
        print i, v
```

To loop over a sequence in sorted order,

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
        print f
```

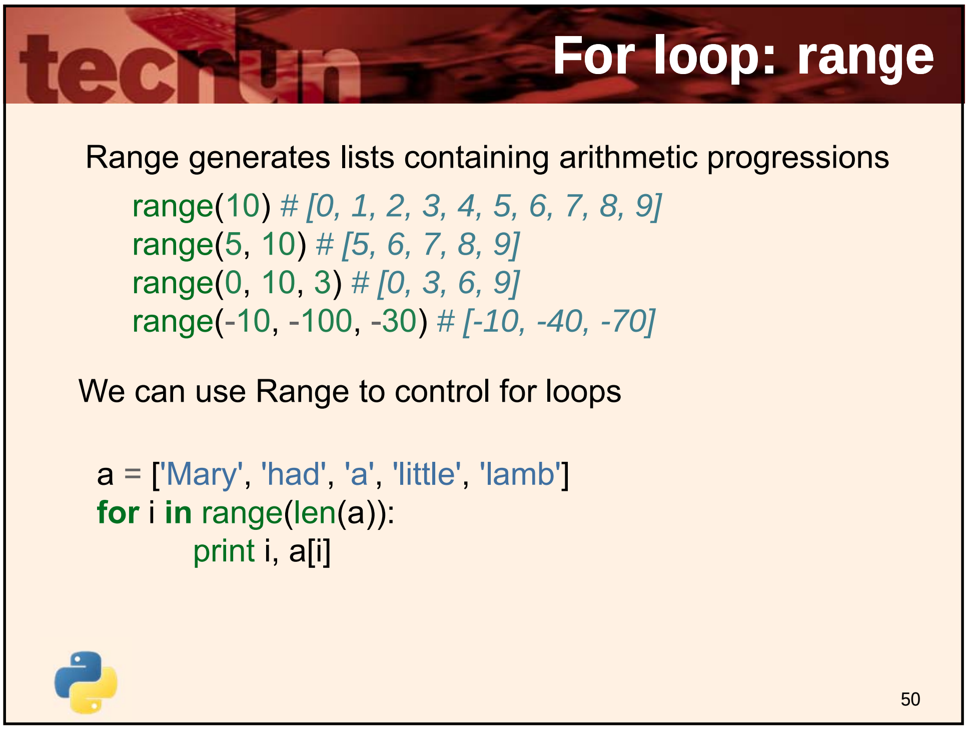
48



For loop: 2 sequences

To loop over two or more sequences at the same time, the entries can be paired with the zip() function

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
        print 'What is your {0}? It is {1}.'.format(q, a)

# Loop →What is your name? It is lancelot.
# Loop → What is your quest? It is the holy grail.
# Loop → What is your favorite color? It is blue.
```



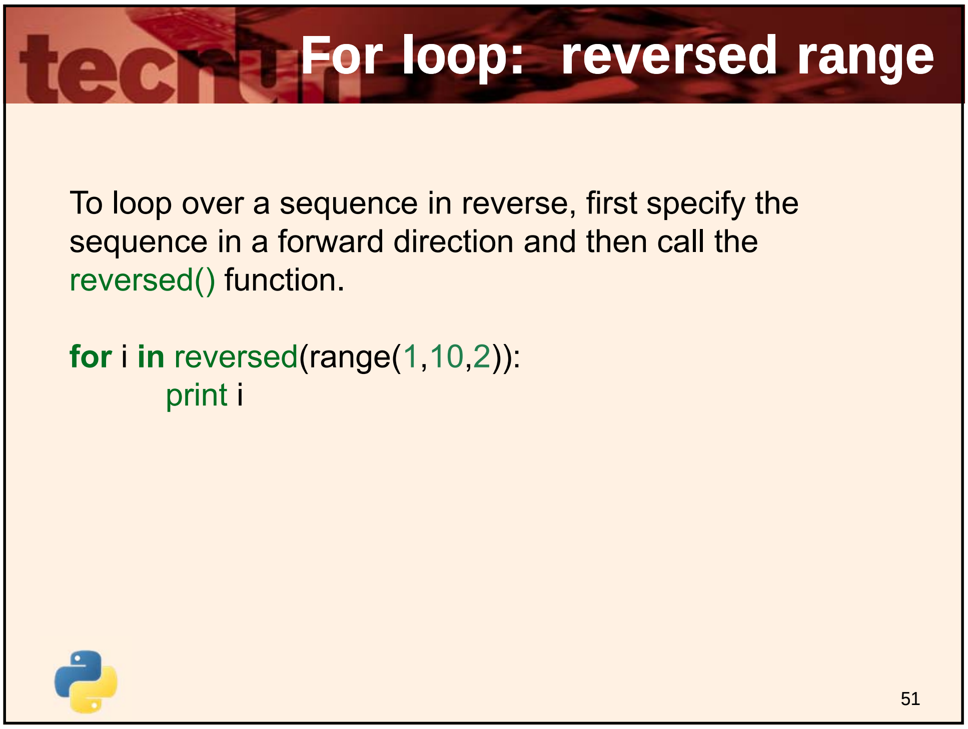
For loop: range

Range generates lists containing arithmetic progressions

```
range(10) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(5, 10) # [5, 6, 7, 8, 9]
range(0, 10, 3) # [0, 3, 6, 9]
range(-10, -100, -30) # [-10, -40, -70]
```

We can use Range to control for loops

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
        print i, a[i]
```

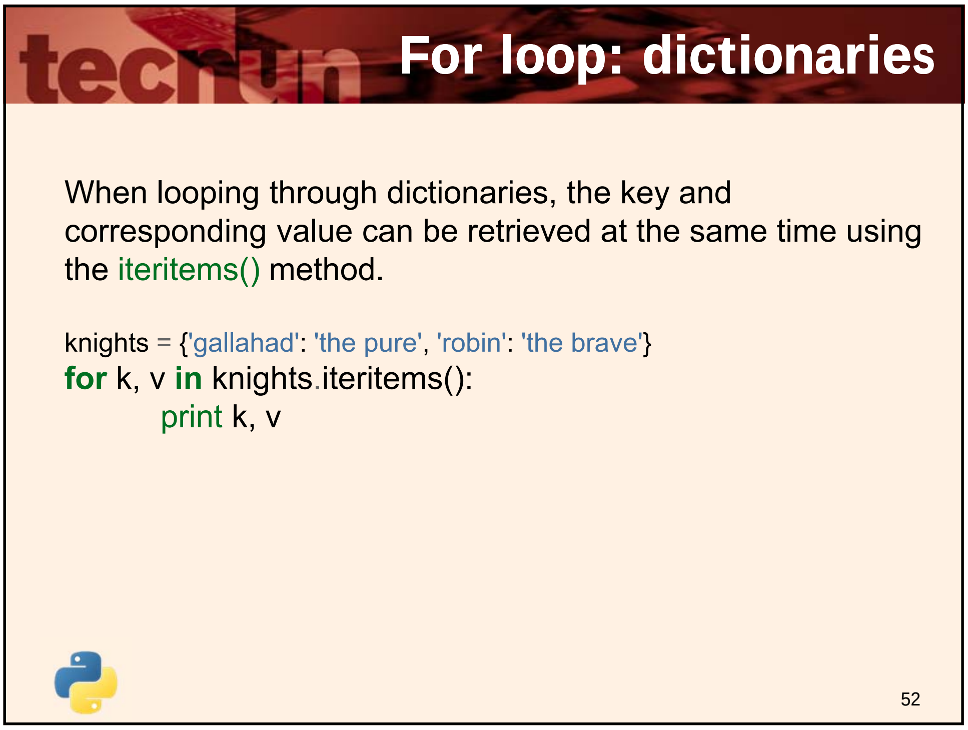


For loop: reversed range

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
for i in reversed(range(1,10,2)):
        print i
```

51



For loop: dictionaries

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the iteritems() method.

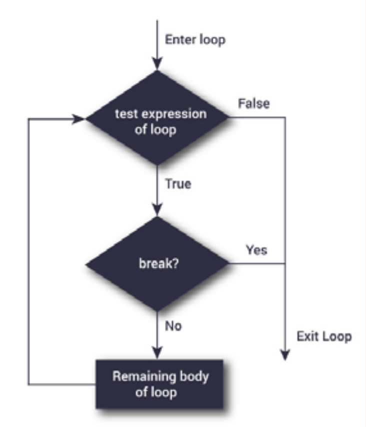
```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.iteritems():
        print k, v
```

52

Break in loops

break: like C→ breaks out the innermost loop (**for/while**)

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print n, 'equals', x, '*', n/x
            break
```

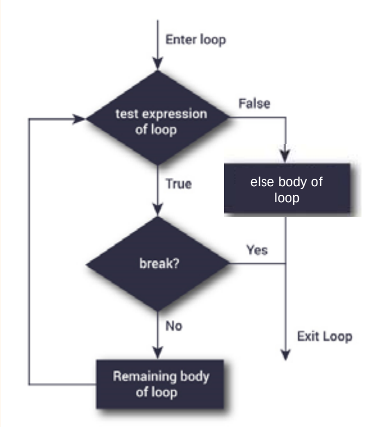


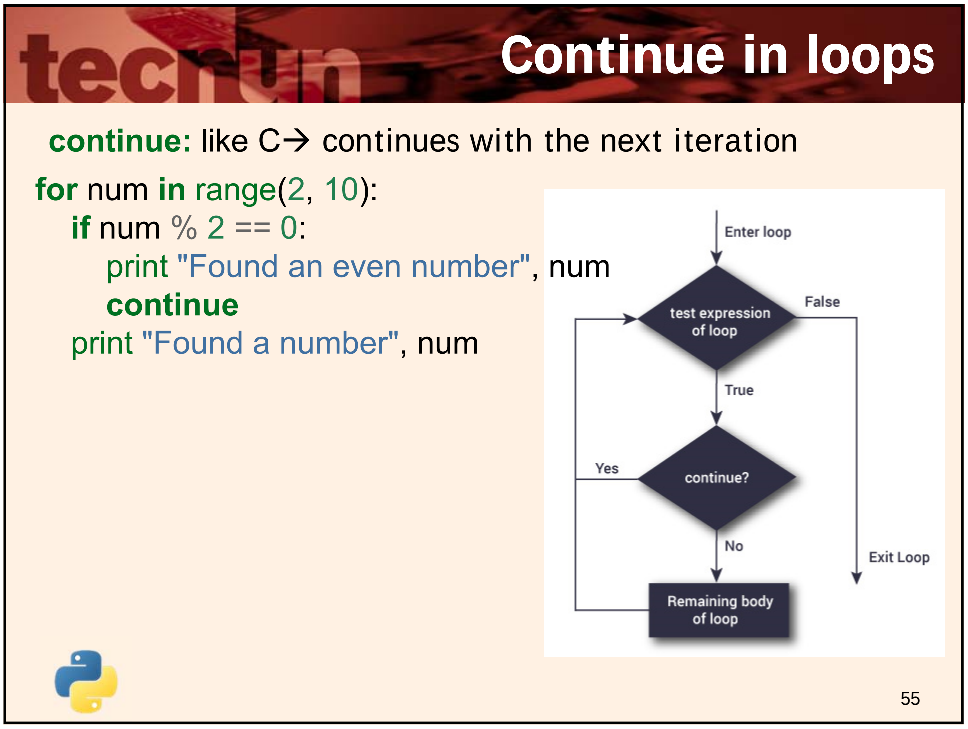
Else in loops

else: ; executed when the loop terminates

- **for**→through exhaustion of the list
- **while**→condition becomes false
- not when the loop is terminated by a **break**

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print n, 'equals', x, '*', n/x
            break
    else:
    # loop fell through without finding a factor
        print n, 'is a prime number'
```

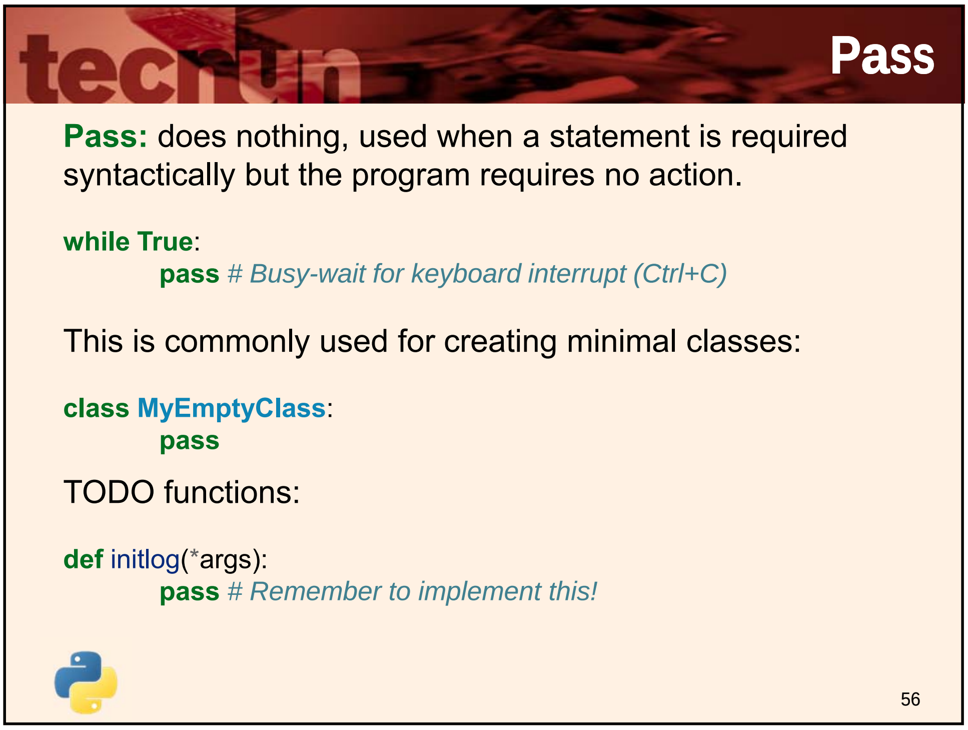




Continue in loops

continue: like C→ continues with the next iteration

```
for num in range(2, 10):
    if num % 2 == 0:
        print "Found an even number", num
        continue
    print "Found a number", num
```



Pass

Pass: does nothing, used when a statement is required syntactically but the program requires no action.

```
while True:
        pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

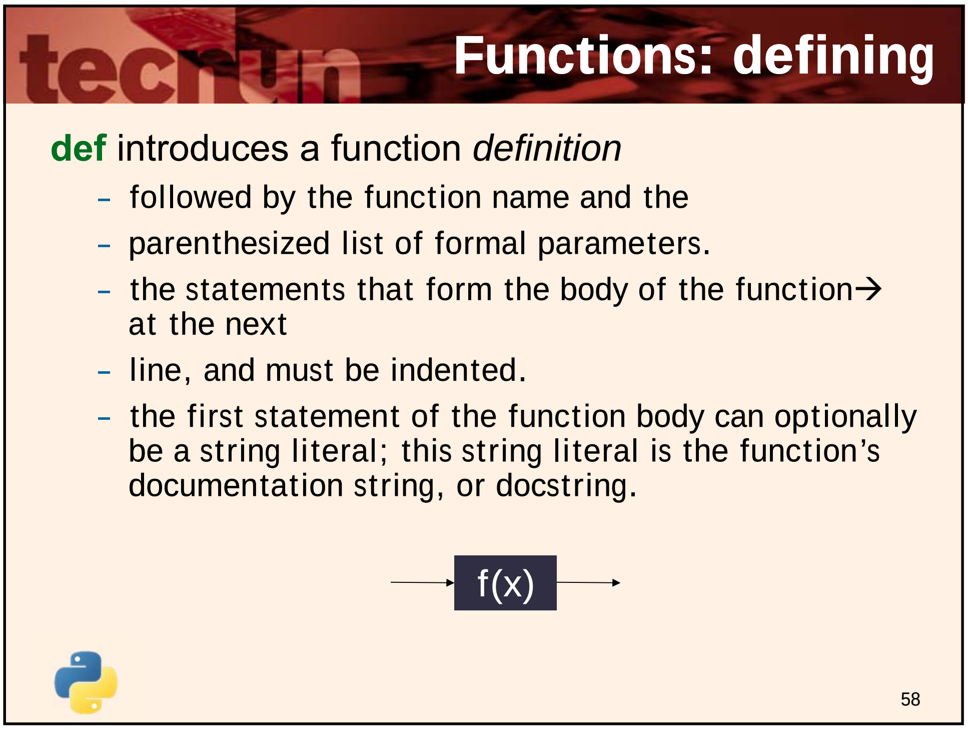
This is commonly used for creating minimal classes:

```
class MyEmptyClass:
        pass
```

TODO functions:

```
def initlog(*args):
        pass # Remember to implement this!
```

Functions in python

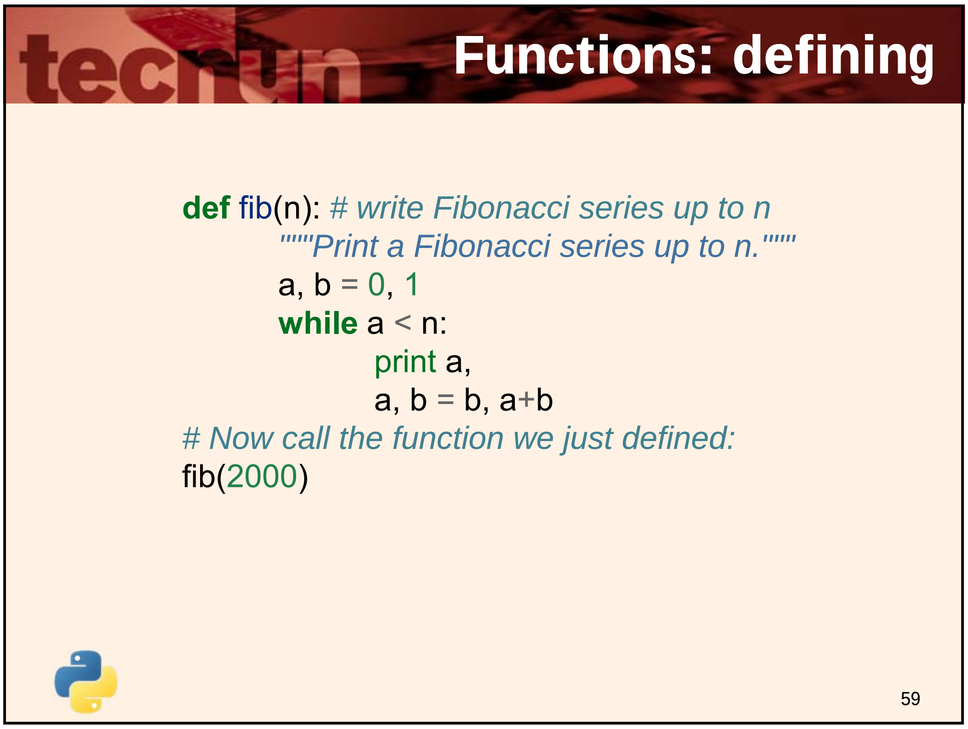


Functions: defining

def introduces a function *definition*

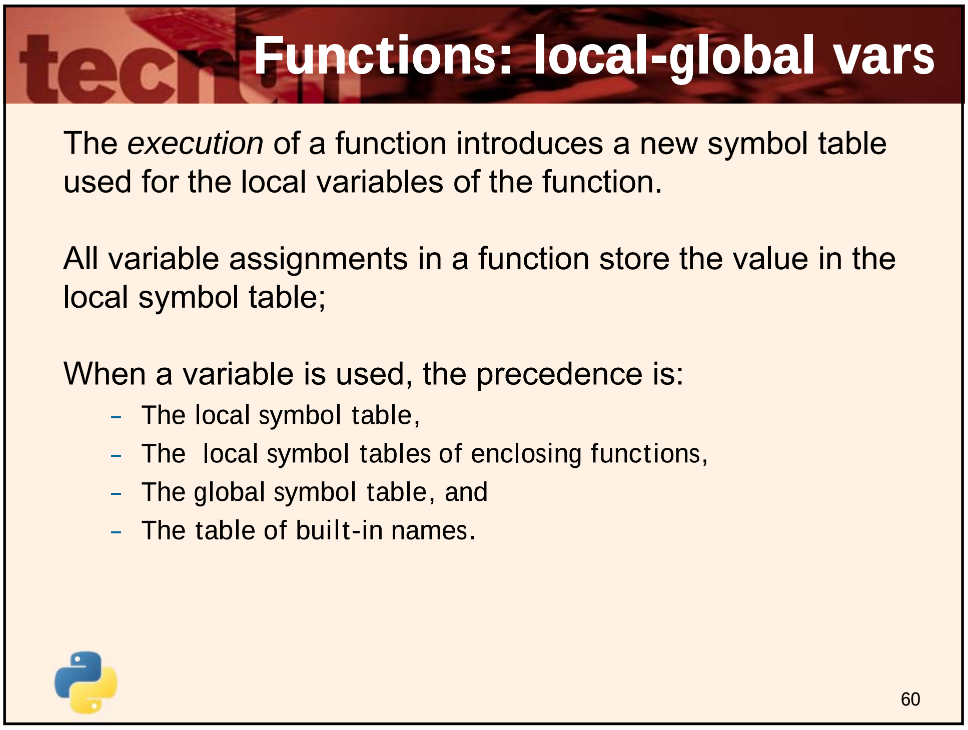
- followed by the function name and the
- parenthesized list of formal parameters.
- the statements that form the body of the function→ at the next
- line, and must be indented.
- the first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring.

f(x)



Functions: defining

```
def fib(n): # write Fibonacci series up to n
        """Print a Fibonacci series up to n."""
        a, b = 0, 1
        while a < n:
                print a,
                a, b = b, a+b
# Now call the function we just defined:
fib(2000)
```



Functions: local-global vars

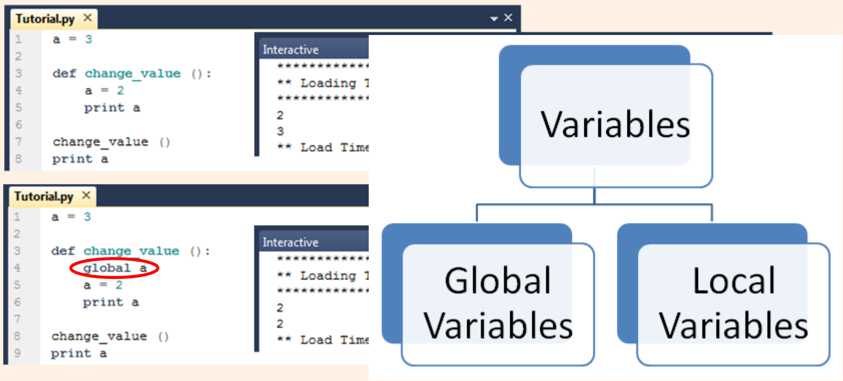
The *execution* of a function introduces a new symbol table used for the local variables of the function.

All variable assignments in a function store the value in the local symbol table;

When a variable is used, the precedence is:

- The local symbol table,
- The local symbol tables of enclosing functions,
- The global symbol table, and
- The table of built-in names.

Funtions: Scope of variables



Tutorial.py

```
a = 3

def change_value ():
    a = 2
    print a

change_value ()
print a
```

Interactive

```
************
** Loading T
************
2
3
** Load Time
```

Tutorial.py

```
a = 3

def change_value ():
    global a
    a = 2
    print a

change_value ()
print a
```

Interactive

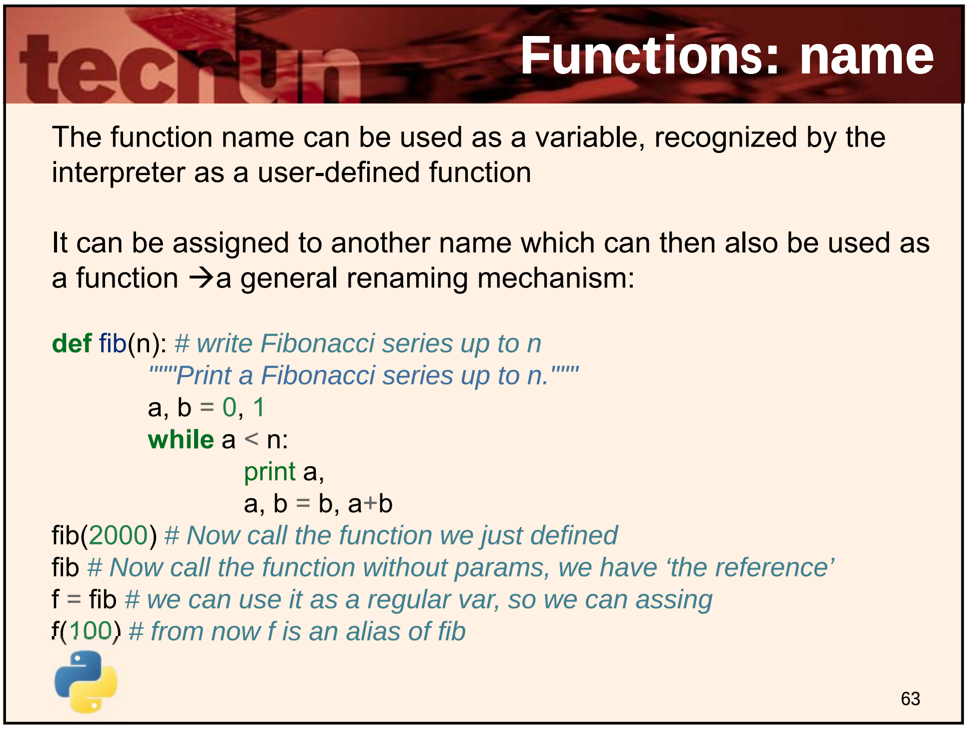
```
************
** Loading T
************
2
2
** Load Time
```

- **GLOBALS** available from any place of the script
- **LOCALS** only inside the actual function

Functions: scope of input params

The actual parameters (arguments) to a function call are introduced in the local symbol table.

The arguments are passed using *call by value.* That is to say a local variable which value is equal to the value of the variable used as parameter when the function was called (like in C)



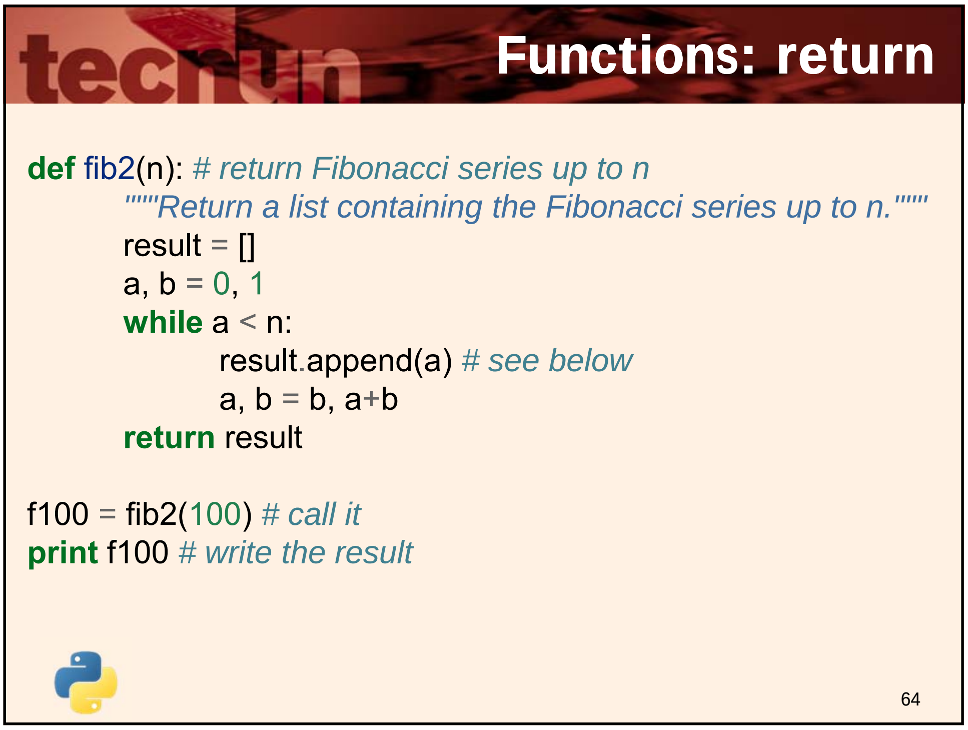
Functions: name

The function name can be used as a variable, recognized by the interpreter as a user-defined function

It can be assigned to another name which can then also be used as a function →a general renaming mechanism:

```
def fib(n): # write Fibonacci series up to n
        """Print a Fibonacci series up to n."""
        a, b = 0, 1
        while a < n:
                print a,
                a, b = b, a+b
fib(2000) # Now call the function we just defined
fib # Now call the function without params, we have ‘the reference’
f = fib # we can use it as a regular var, so we can assing
f(100) # from now f is an alias of fib
```

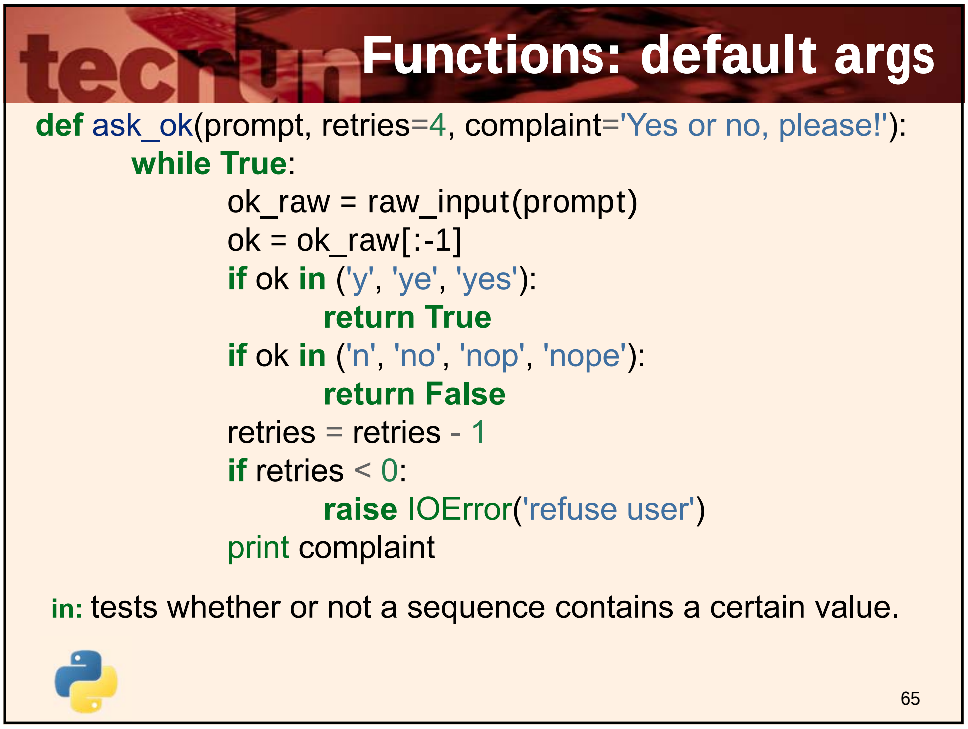
63



Functions: return

```
def fib2(n): # return Fibonacci series up to n
        """Return a list containing the Fibonacci series up to n."""
        result = []
        a, b = 0, 1
        while a < n:
                result.append(a) # see below
                a, b = b, a+b
        return result

f100 = fib2(100) # call it
print f100 # write the result
```

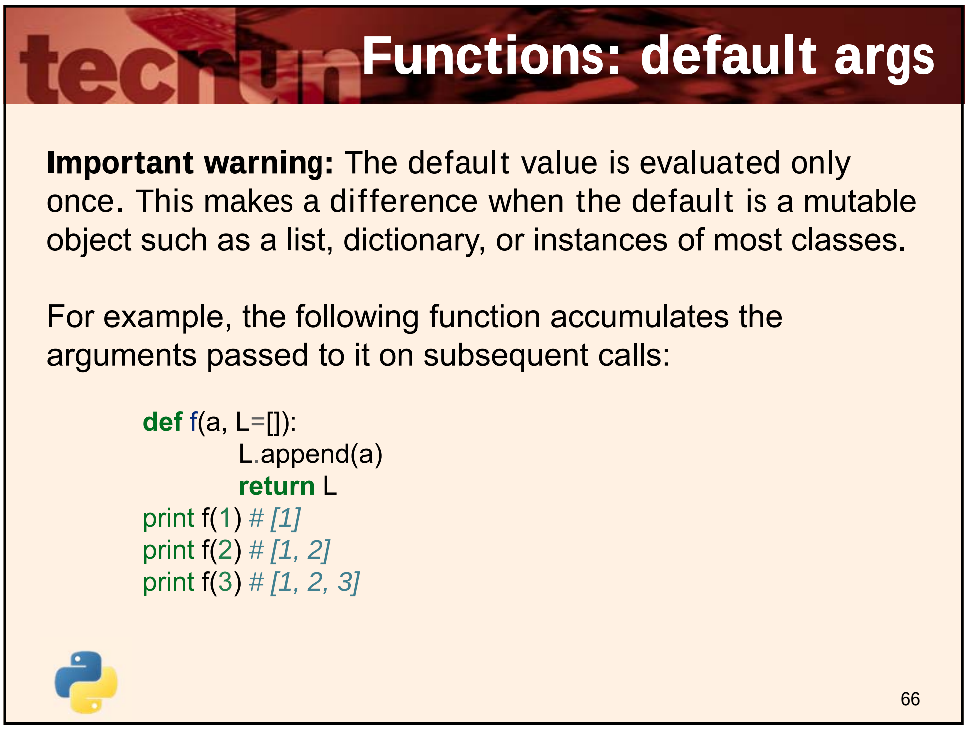
64



Functions: default args

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok_raw = raw_input(prompt)
        ok = ok_raw[:-1]
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refuse user')
        print complaint
```

in: tests whether or not a sequence contains a certain value.

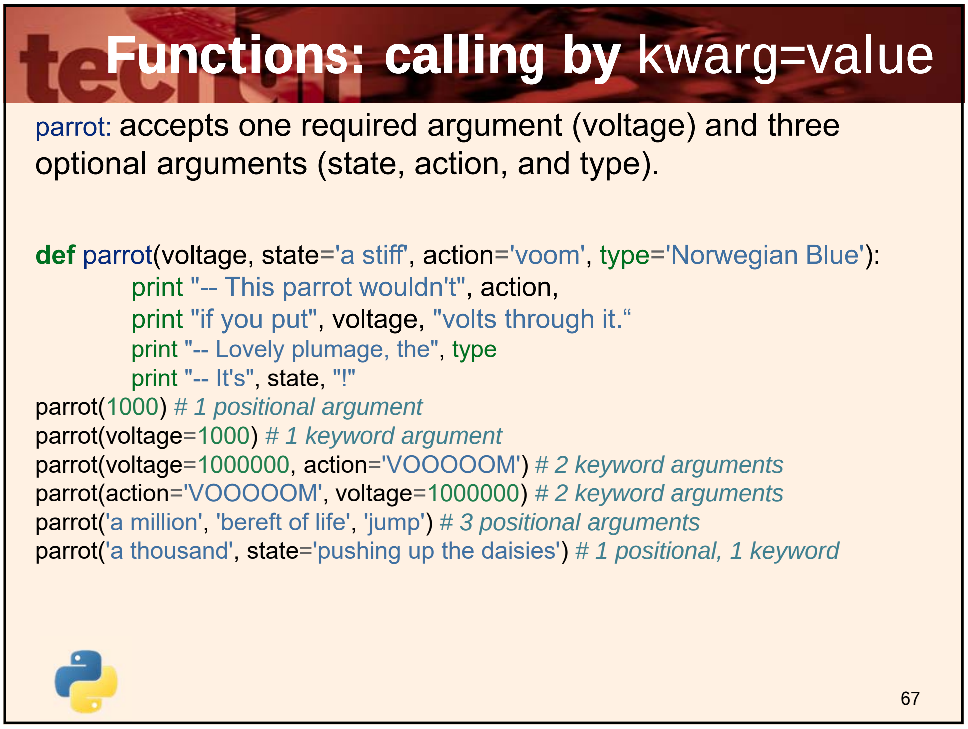


Functions: default args

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

For example, the following function accumulates the arguments passed to it on subsequent calls:

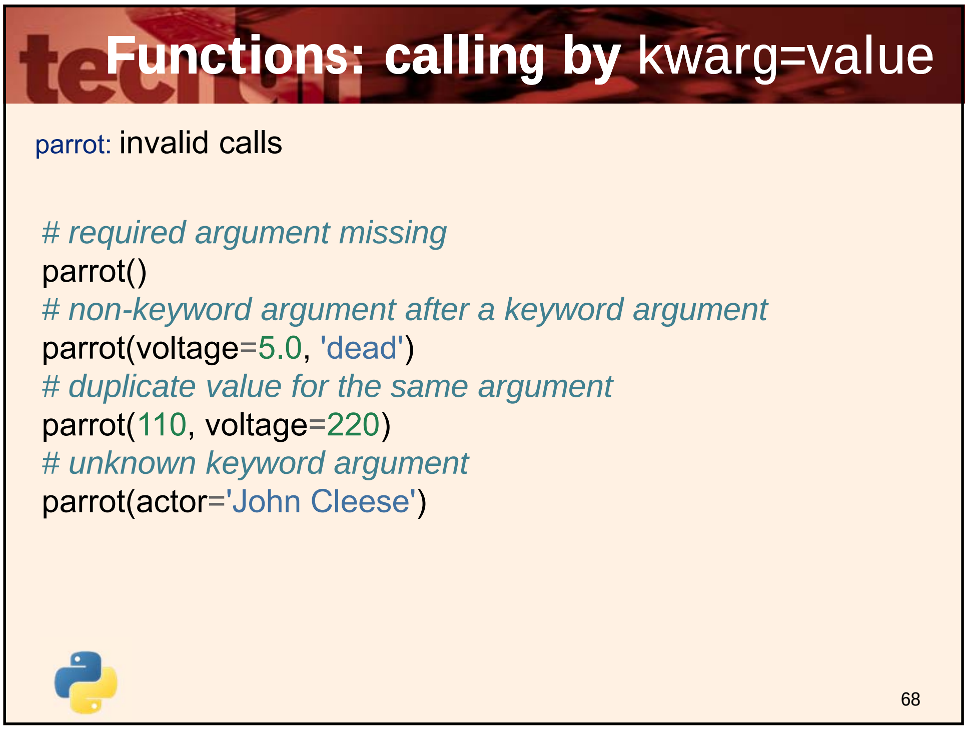
```
def f(a, L=[]):
    L.append(a)
    return L
print f(1) # [1]
print f(2) # [1, 2]
print f(3) # [1, 2, 3]
```



Functions: calling by kwarg=value

parrot: accepts one required argument (voltage) and three optional arguments (state, action, and type).

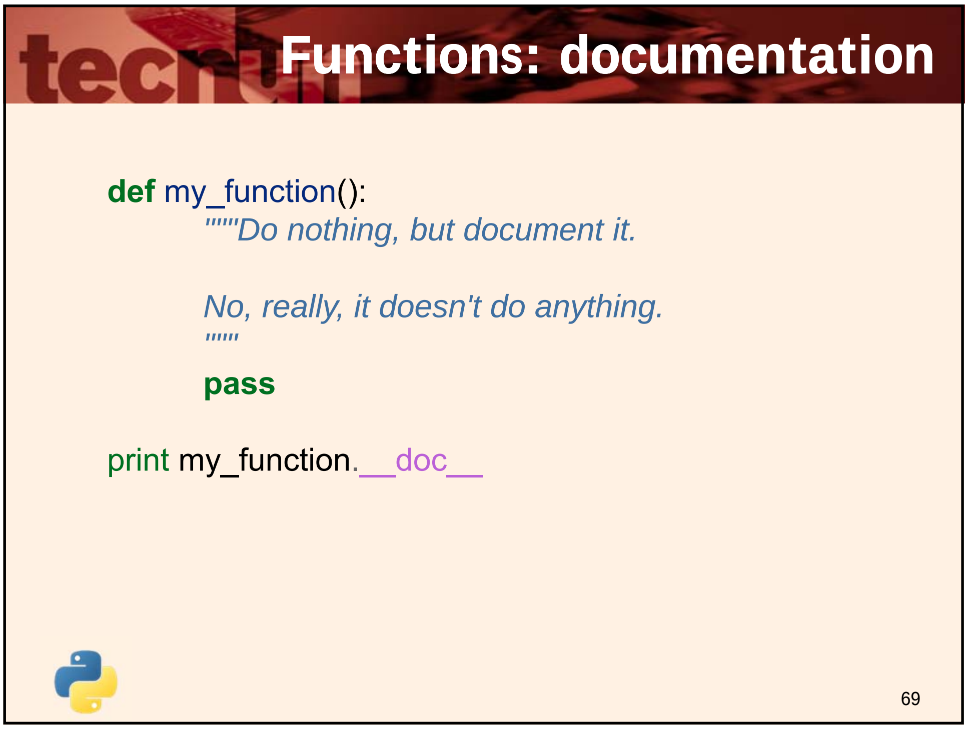
```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
        print "-- This parrot wouldn't", action,
        print "if you put", voltage, "volts through it.“
        print "-- Lovely plumage, the", type
        print "-- It's", state, "!"
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```



Functions: calling by kwarg=value

parrot: invalid calls

```
# required argument missing
parrot()
# non-keyword argument after a keyword argument
parrot(voltage=5.0, 'dead')
# duplicate value for the same argument
parrot(110, voltage=220)
# unknown keyword argument
parrot(actor='John Cleese')
```

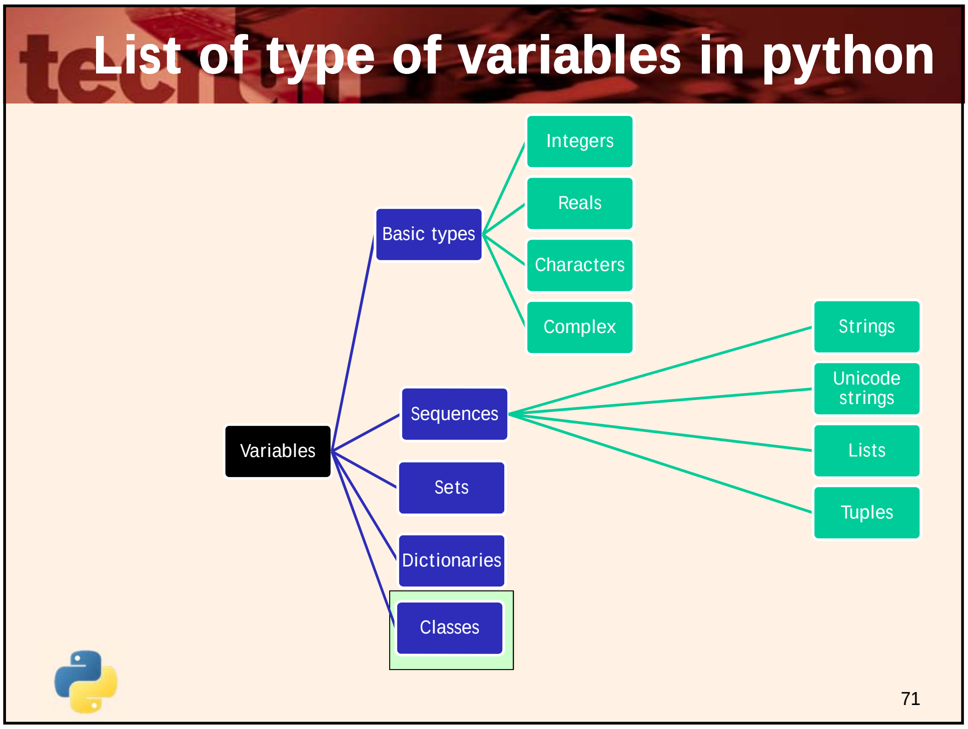


Functions: documentation

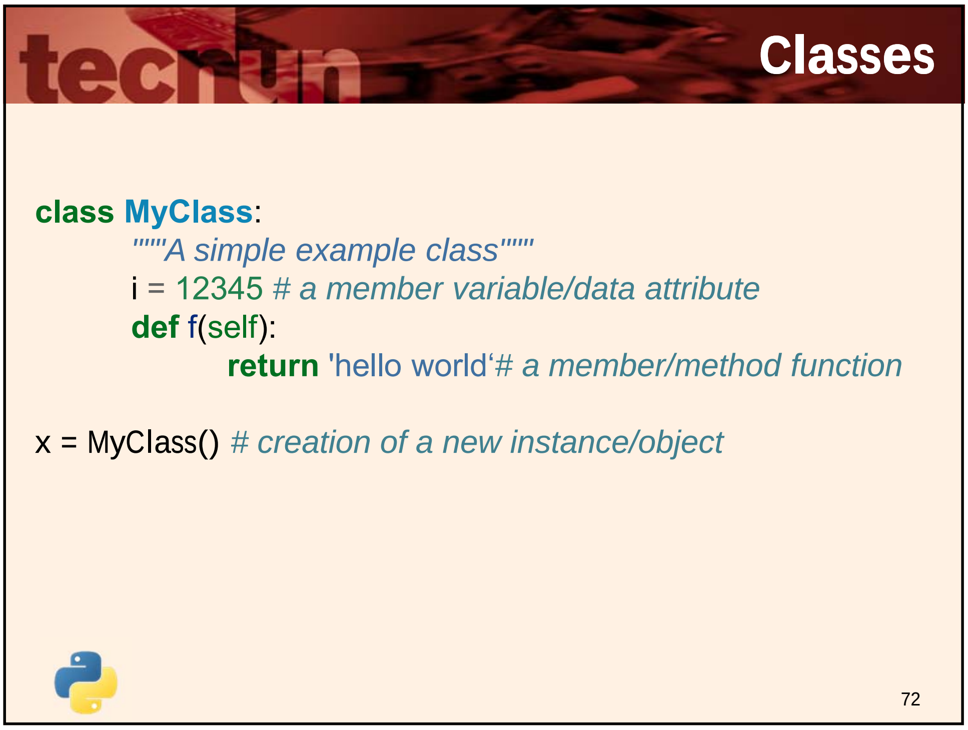
```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass

print my_function.__doc__
```

69

Classes in python

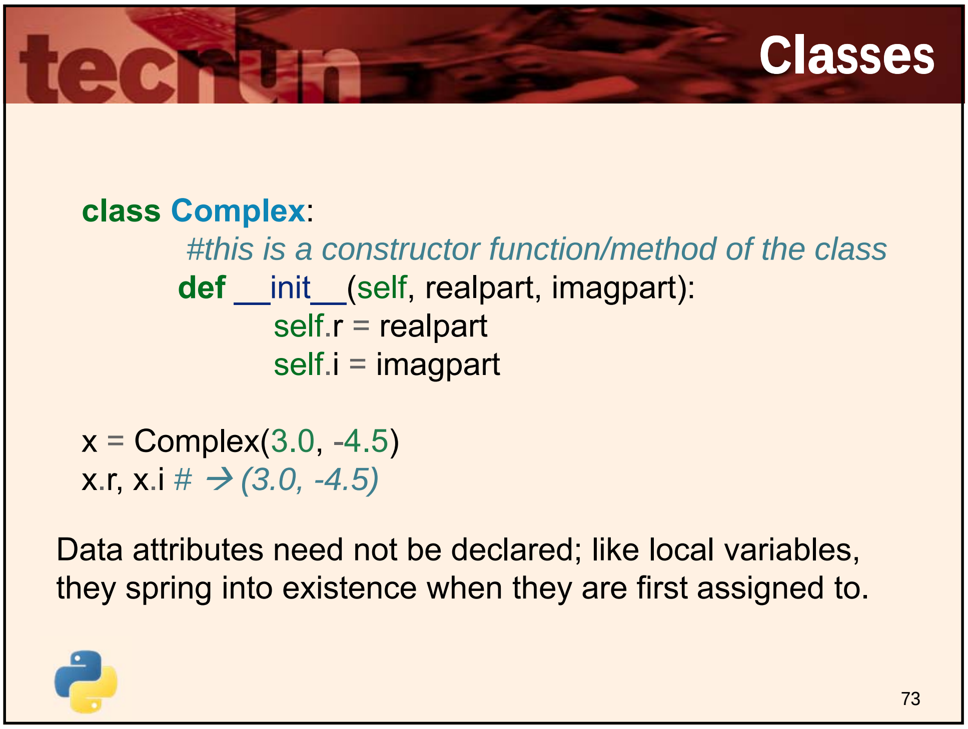


List of type of variables in python



Classes

```
class MyClass:
        """A simple example class"""
        i = 12345 # a member variable/data attribute
        def f(self):
                return 'hello world‘# a member/method function

x = MyClass() # creation of a new instance/object
```

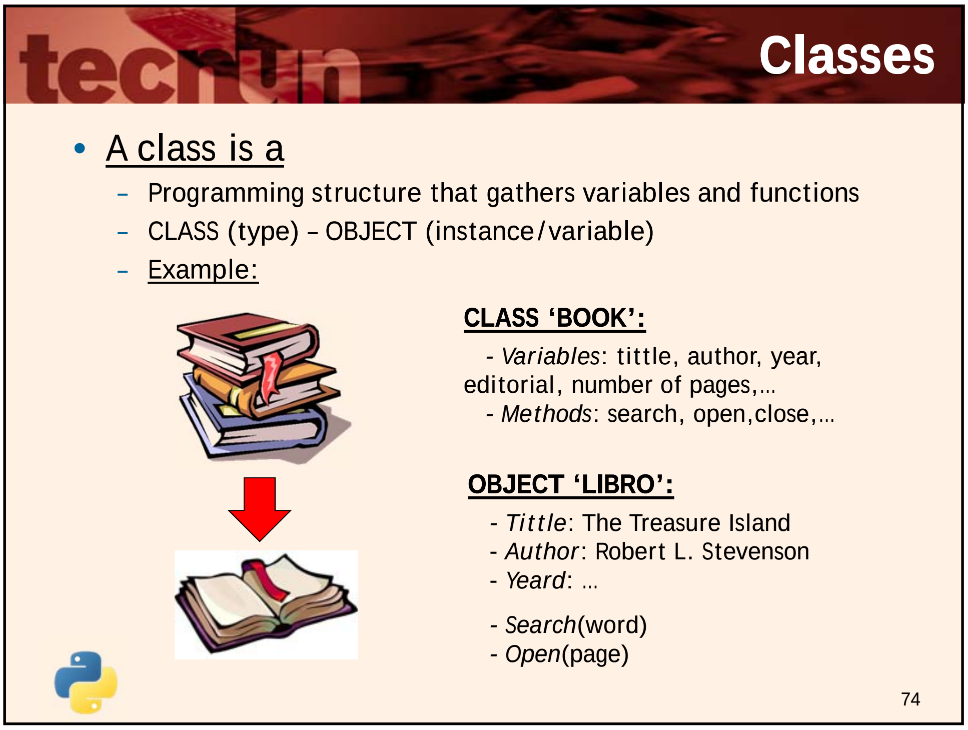


Classes

```
class Complex:
          #this is a constructor function/method of the class
          def __init__(self, realpart, imagpart):
                    self.r = realpart
                    self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i # → (3.0, -4.5)
```

Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

73



Classes

- <u>A class is a</u>
  - Programming structure that gathers variables and functions
  - CLASS (type) – OBJECT (instance/variable)
  - <u>Example:</u>

CLASS 'BOOK':

- *Variables*: tittle, author, year, editorial, number of pages,…
- *Methods*: search, open,close,…

OBJECT 'LIBRO':

- *Tittle*: The Treasure Island
- *Author*: Robert L. Stevenson
- *Yeard*: …

- *Search*(word)
- *Open*(page)

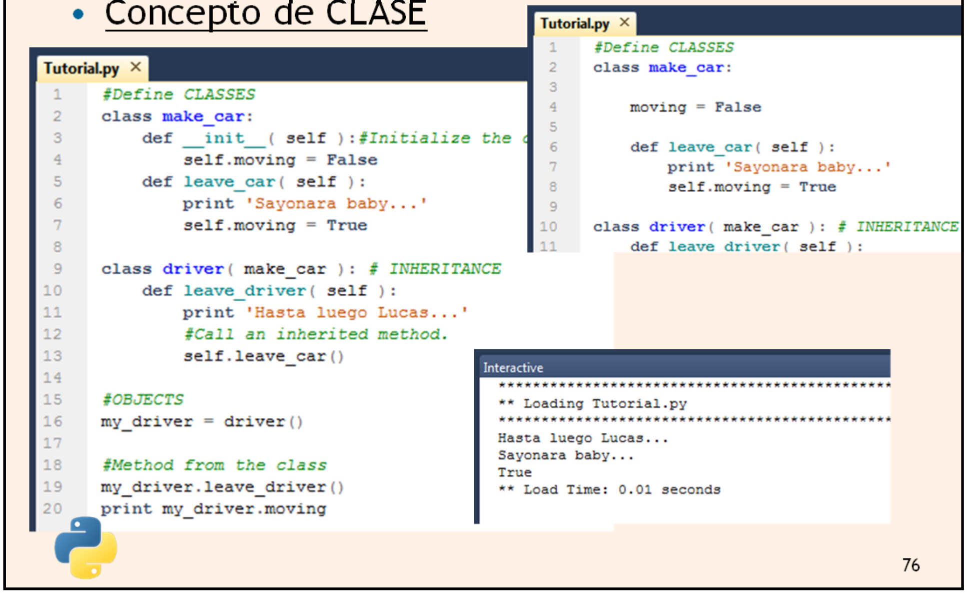
74

Classes

```
class Dog:
        kind = 'canine' # class variable shared by all instances

        def __init__(self, name):
                # instance variable unique to each instance
                self.name = name

d = Dog('Fido')
e = Dog('Buddy')
print d.kind # shared by all dogs → 'canine'
print e.kind # shared by all dogs → 'canine'
print d.name # unique to d → 'Fido'
print e.name # unique to e → 'Buddy'
```

Classes

- <u>Concepto de CLASE</u>

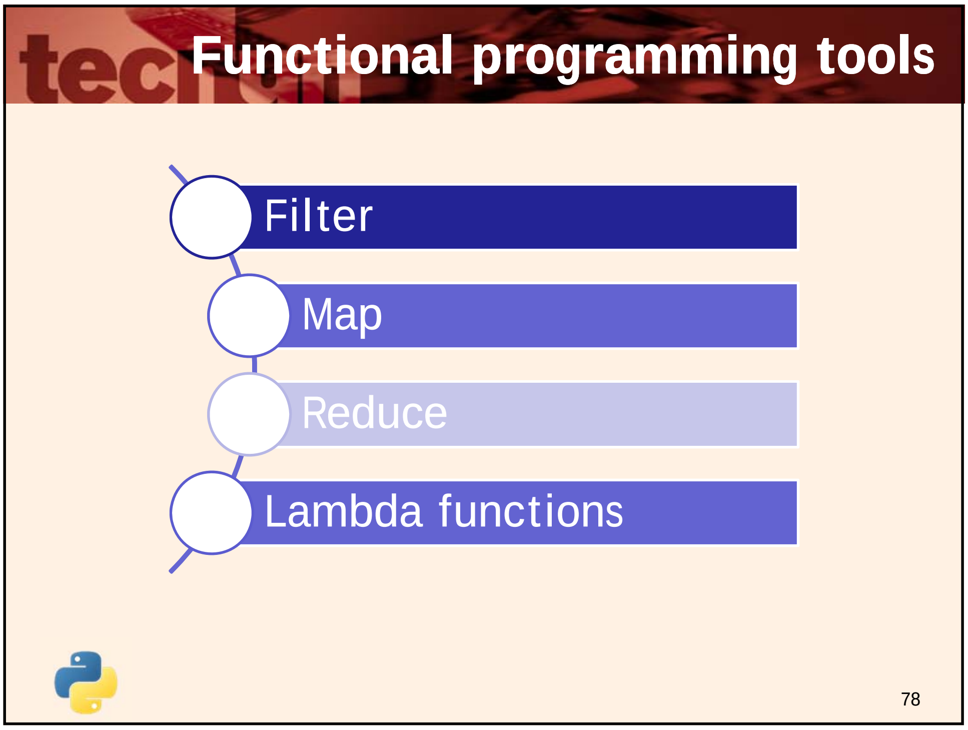


```
#Define CLASSES
class make_car:
    def __init__( self ):#Initialize the
        self.moving = False
    def leave_car( self ):
        print 'Sayonara baby...'
        self.moving = True

class driver( make_car ): # INHERITANCE
    def leave_driver( self ):
        print 'Hasta luego Lucas...'
        #Call an inherited method.
        self.leave_car()

#OBJECTS
my_driver = driver()

#Method from the class
my_driver.leave_driver()
print my_driver.moving
```

```
#Define CLASSES
class make_car:

    moving = False

    def leave_car( self ):
        print 'Sayonara baby...'
        self.moving = True

class driver( make_car ): # INHERITANCE
    def leave_driver( self ):
```

```
************************************************
** Loading Tutorial.py
************************************************
Hasta luego Lucas...
Sayonara baby...
True
** Load Time: 0.01 seconds
```

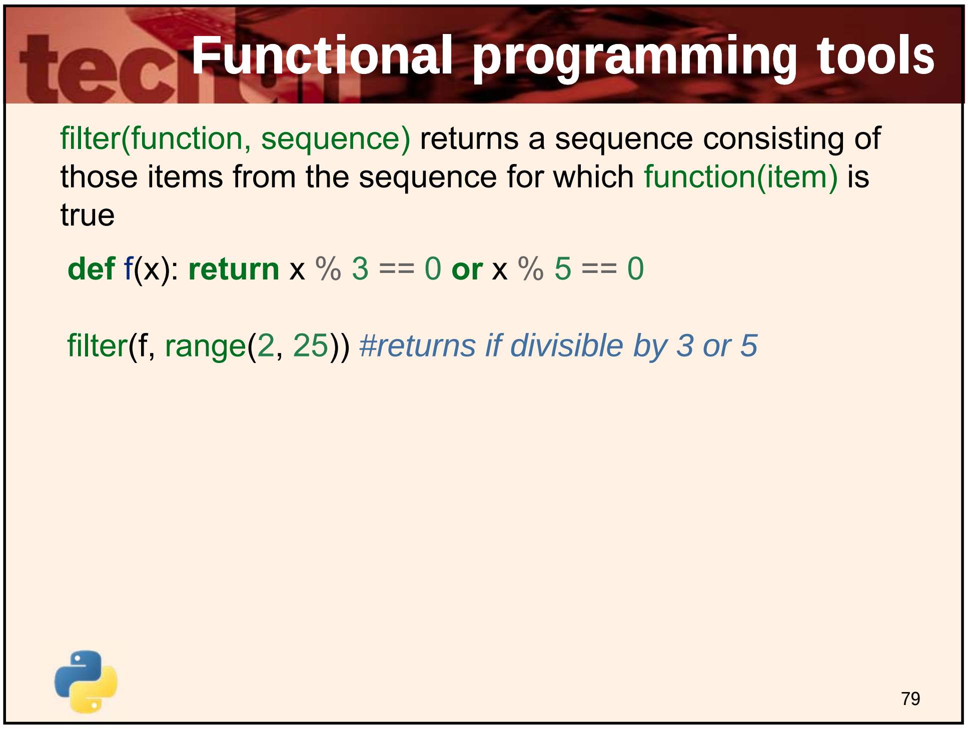
Functional programming tools in python



Functional programming tools

- Filter
- Map
- Reduce
- Lambda functions

78

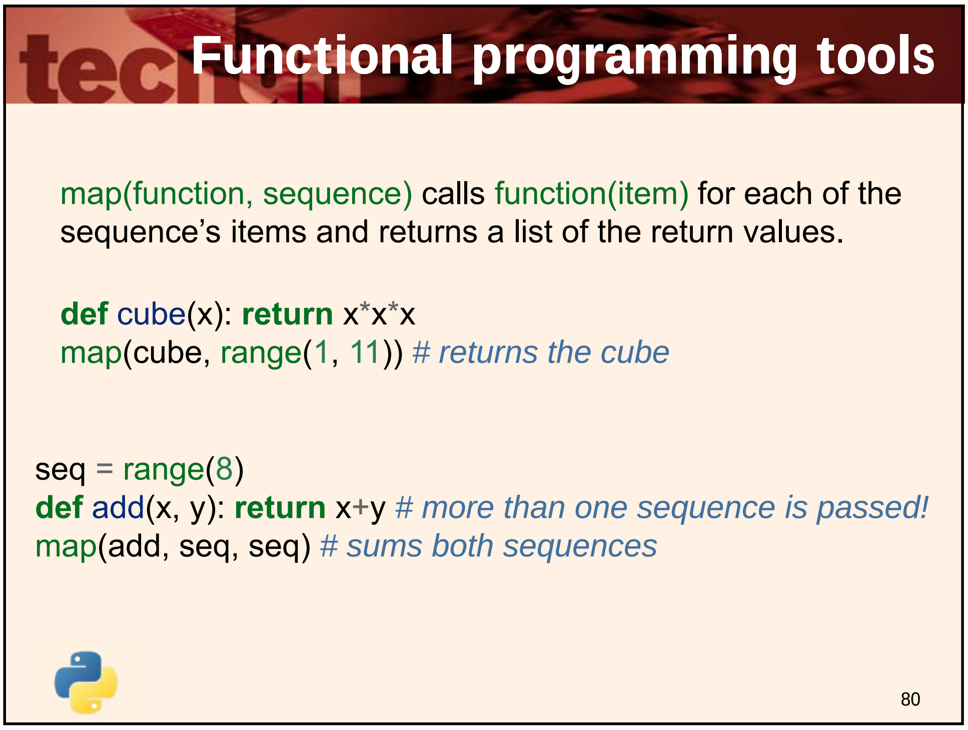


Functional programming tools

filter(function, sequence) returns a sequence consisting of those items from the sequence for which function(item) is true

```
def f(x): return x % 3 == 0 or x % 5 == 0

filter(f, range(2, 25)) #returns if divisible by 3 or 5
```

79

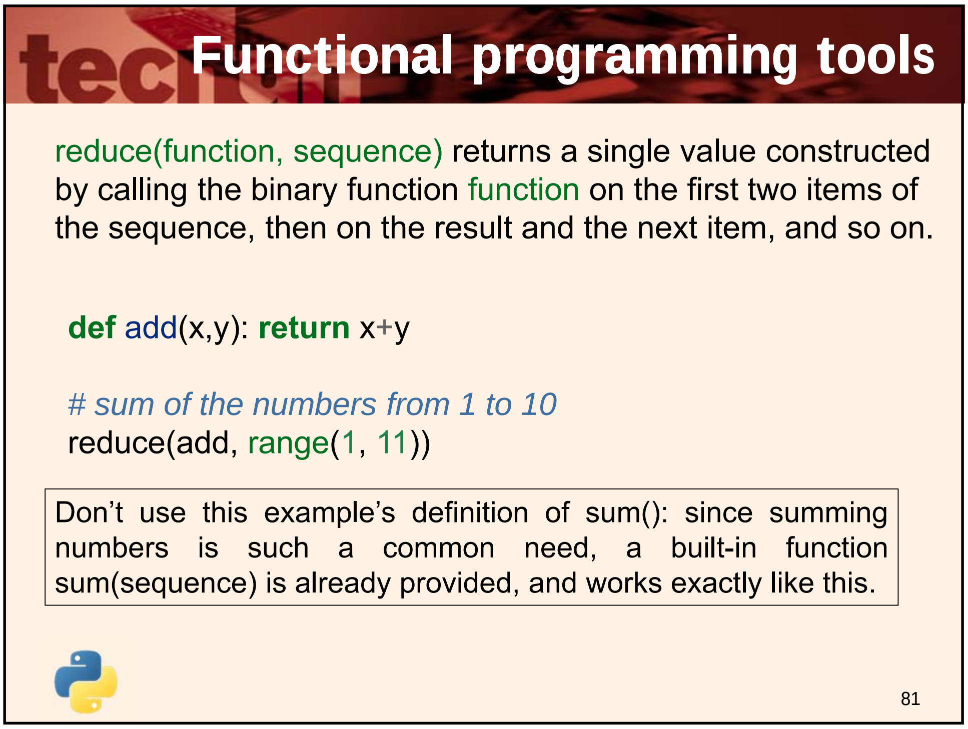


Functional programming tools

map(function, sequence) calls function(item) for each of the sequence’s items and returns a list of the return values.

```
def cube(x): return x*x*x
map(cube, range(1, 11)) # returns the cube

seq = range(8)
def add(x, y): return x+y # more than one sequence is passed!
map(add, seq, seq) # sums both sequences
```

80



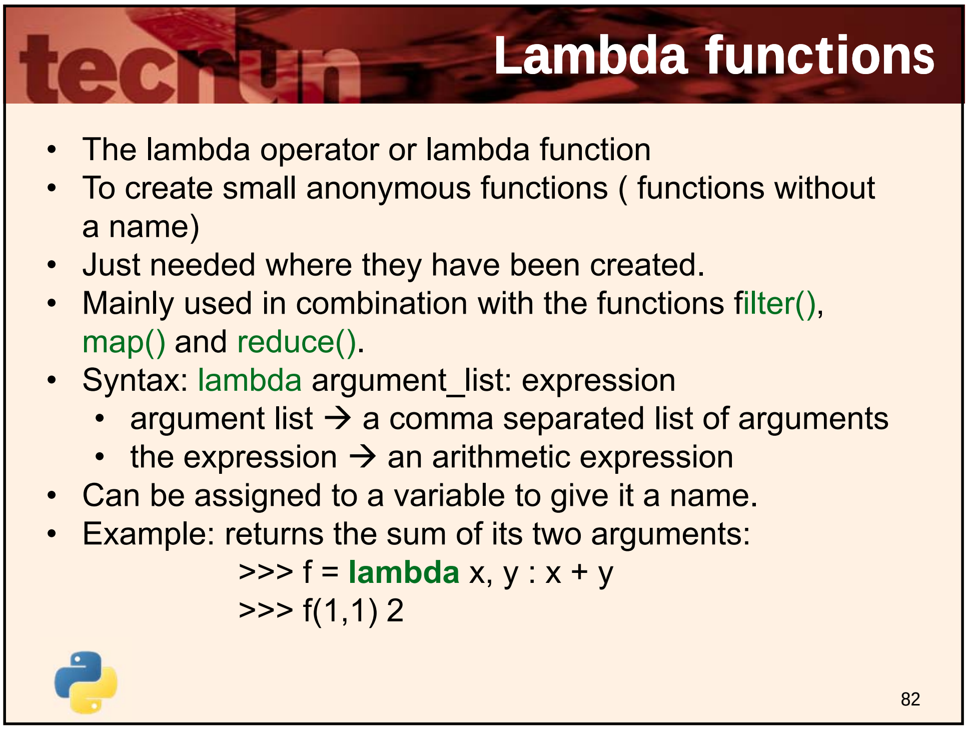
Functional programming tools

reduce(function, sequence) returns a single value constructed by calling the binary function function on the first two items of the sequence, then on the result and the next item, and so on.

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

# sum of the numbers from 1 to 10
reduce(add, range(1, 11))
```

Don’t use this example’s definition of sum(): since summing numbers is such a common need, a built-in function sum(sequence) is already provided, and works exactly like this.

81



Lambda functions

- The lambda operator or lambda function
- To create small anonymous functions (functions without a name)
- Just needed where they have been created.
- Mainly used in combination with the functions filter(), map() and reduce().
- Syntax: lambda argument_list: expression
  - argument list → a comma separated list of arguments
  - the expression → an arithmetic expression
- Can be assigned to a variable to give it a name.
- Example: returns the sum of its two arguments:

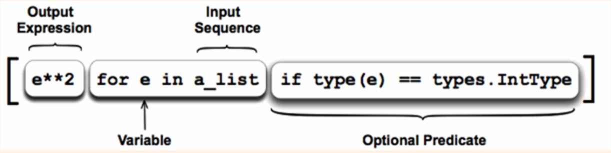
```
>>> f = lambda x, y : x + y
>>> f(1,1) 2
```

82

Comprehensions

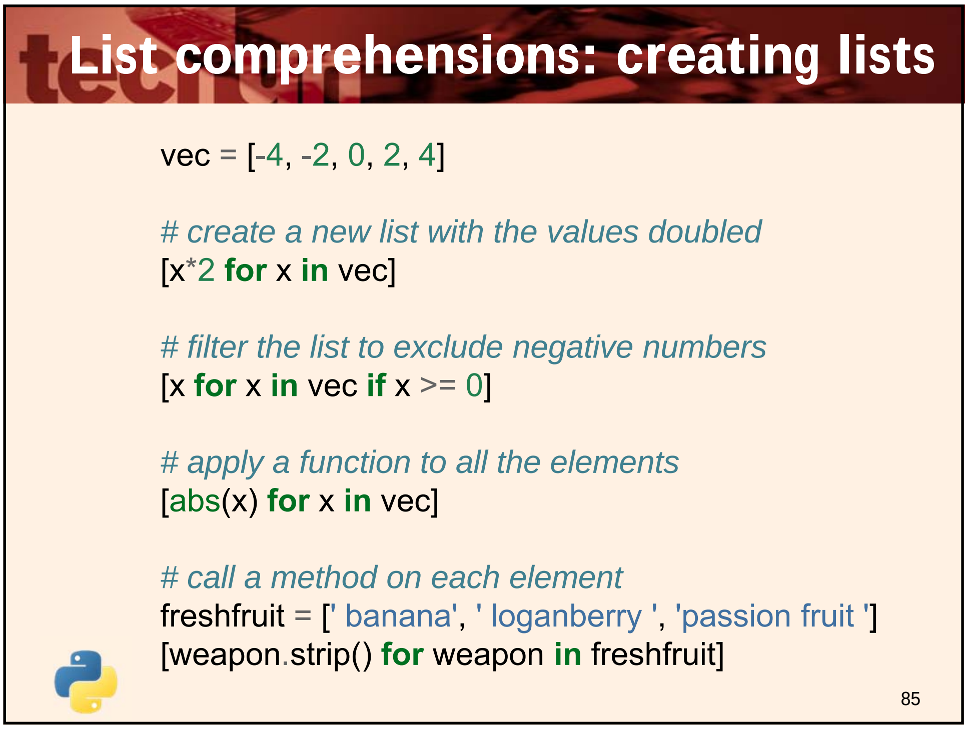
List comprehensions

- A list comprehension consists of the following parts:
  - An Input Sequence.
  - A Variable representing members of the input sequence.
  - An Optional Predicate expression.
  - An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.
- a list of all the integers in a sequence and then square them:



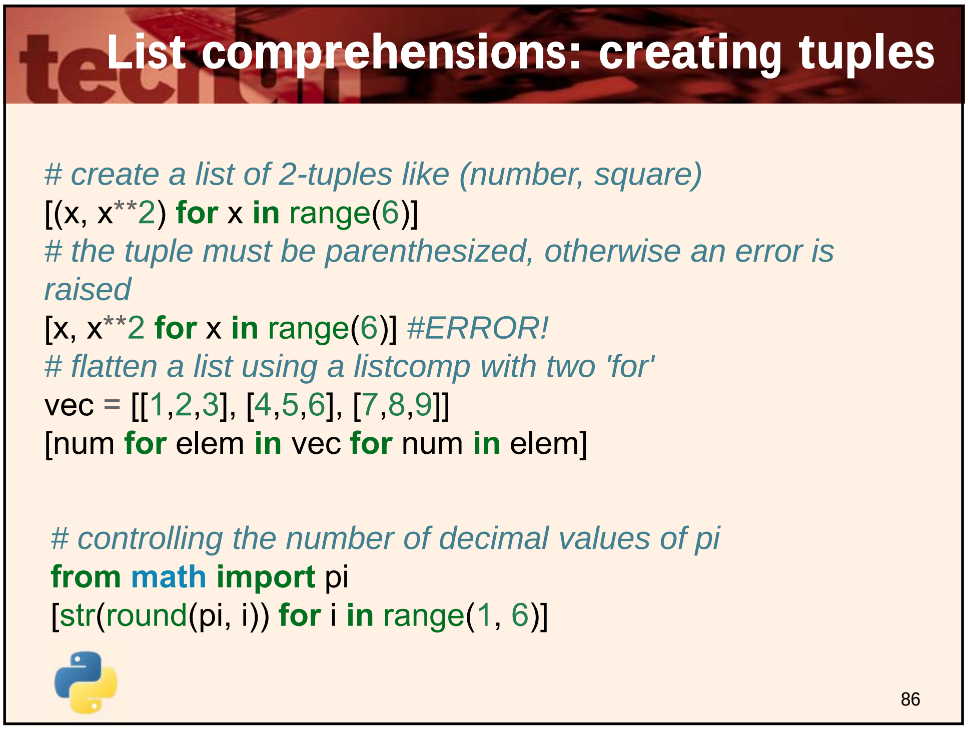
```
[e**2 for e in a_list if type(e) == types.IntType]
```

84



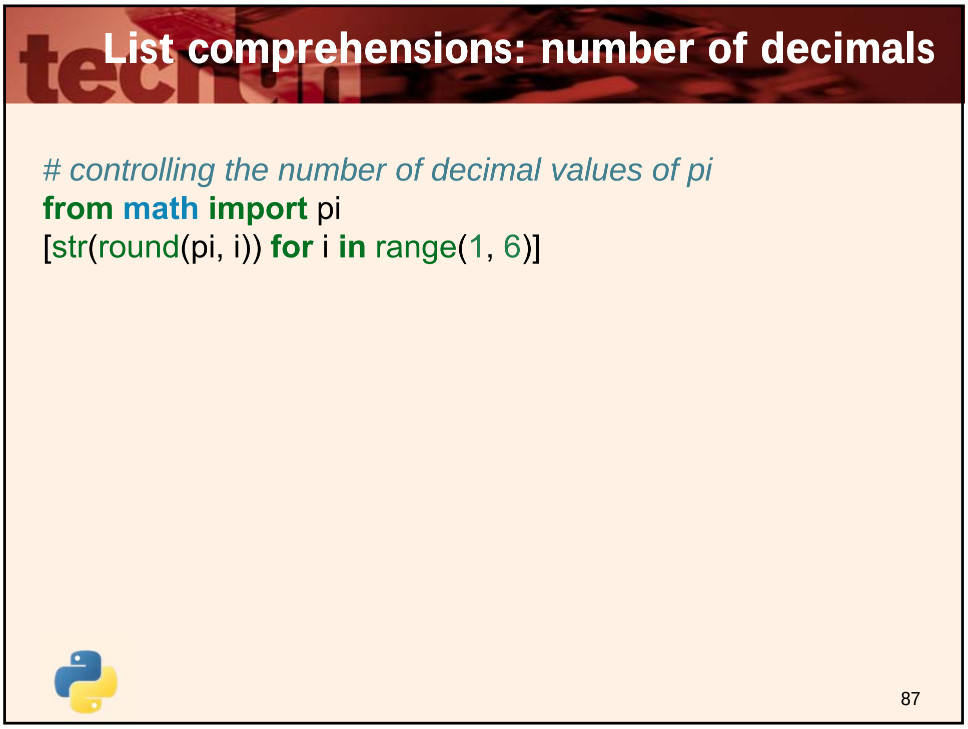
List comprehensions: creating lists

```
vec = [-4, -2, 0, 2, 4]

# create a new list with the values doubled
[x*2 for x in vec]

# filter the list to exclude negative numbers
[x for x in vec if x >= 0]

# apply a function to all the elements
[abs(x) for x in vec]

# call a method on each element
freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
[weapon.strip() for weapon in freshfruit]
```



List comprehensions: creating tuples

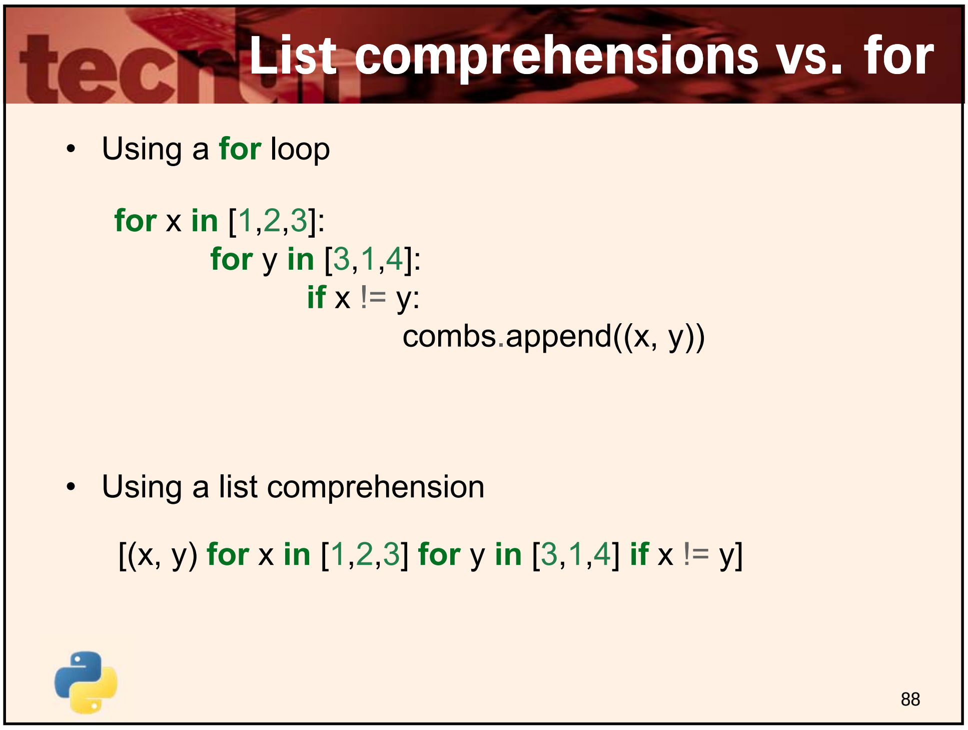
```
# create a list of 2-tuples like (number, square)
[(x, x**2) for x in range(6)]
# the tuple must be parenthesized, otherwise an error is
raised
[x, x**2 for x in range(6)] #ERROR!
# flatten a list using a listcomp with two 'for'
vec = [[1,2,3], [4,5,6], [7,8,9]]
[num for elem in vec for num in elem]


# controlling the number of decimal values of pi
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
```



List comprehensions: number of decimals

```
# controlling the number of decimal values of pi
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
```

87



List comprehensions vs. for

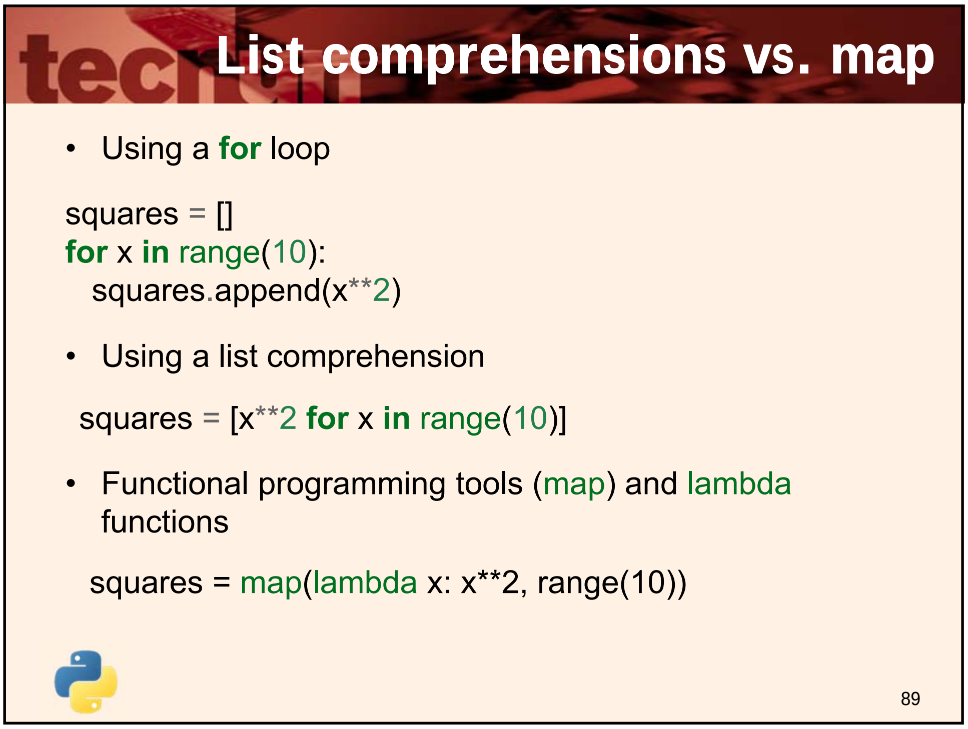
- Using a **for** loop

```
for x in [1,2,3]:
        for y in [3,1,4]:
                if x != y:
                        combs.append((x, y))
```

- Using a list comprehension

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

88



List comprehensions vs. map

- Using a **for** loop

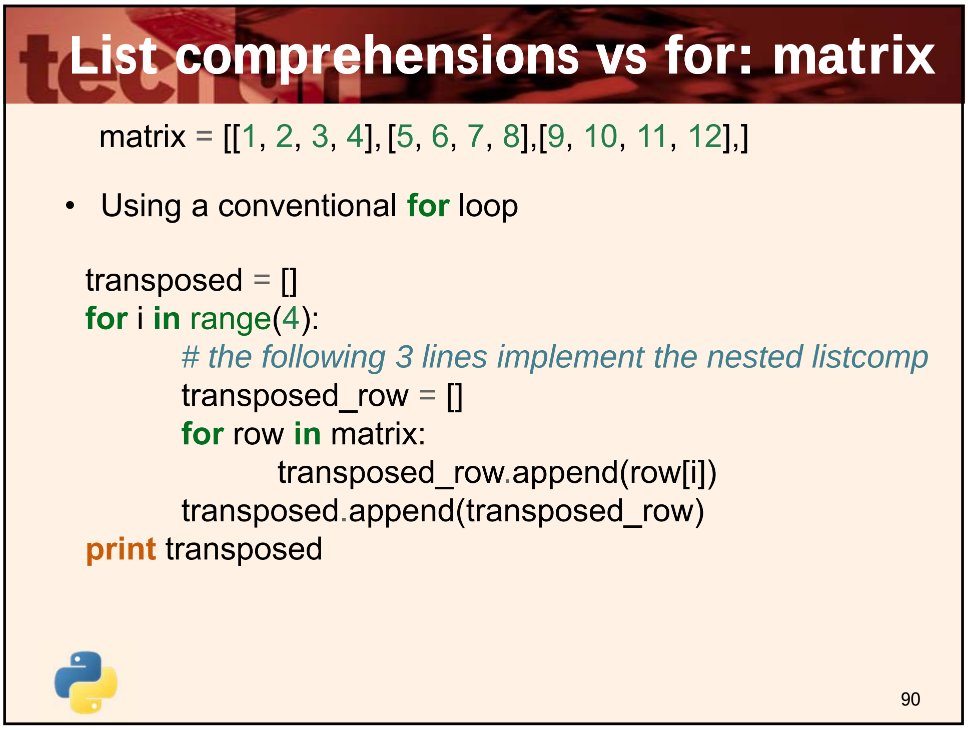
```
squares = []
for x in range(10):
   squares.append(x**2)
```

- Using a list comprehension

```
squares = [x**2 for x in range(10)]
```

- Functional programming tools (map) and lambda functions

```
squares = map(lambda x: x**2, range(10))
```

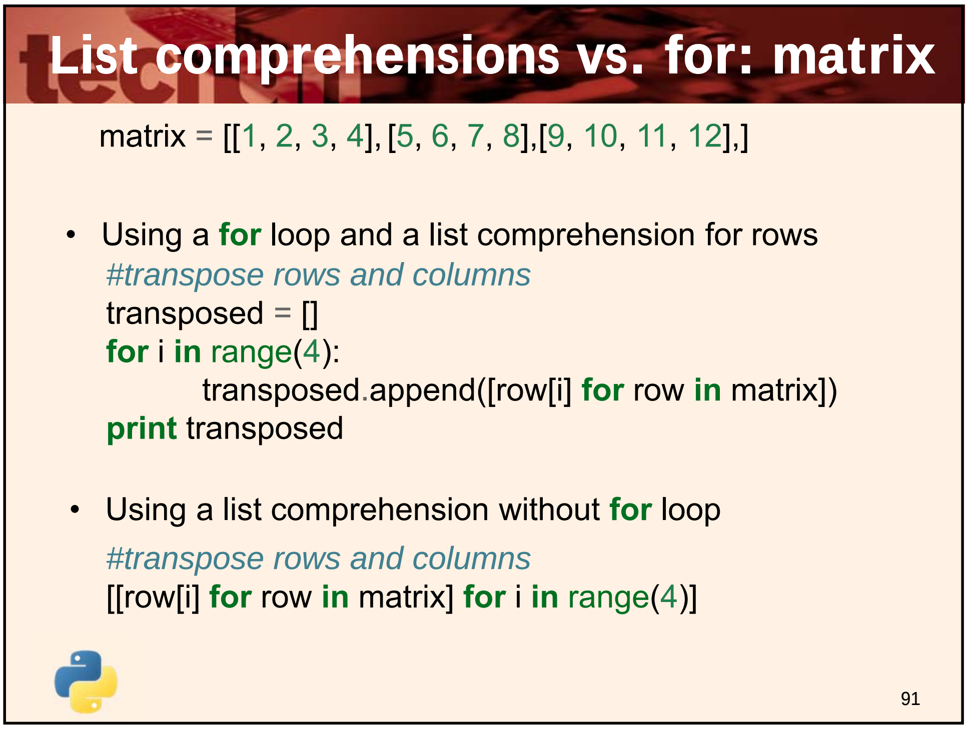


List comprehensions vs for: matrix

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12],]
```

- Using a conventional **for** loop

```
transposed = []
for i in range(4):
        # the following 3 lines implement the nested listcomp
        transposed_row = []
        for row in matrix:
                transposed_row.append(row[i])
        transposed.append(transposed_row)
print transposed
```



List comprehensions vs. for: matrix

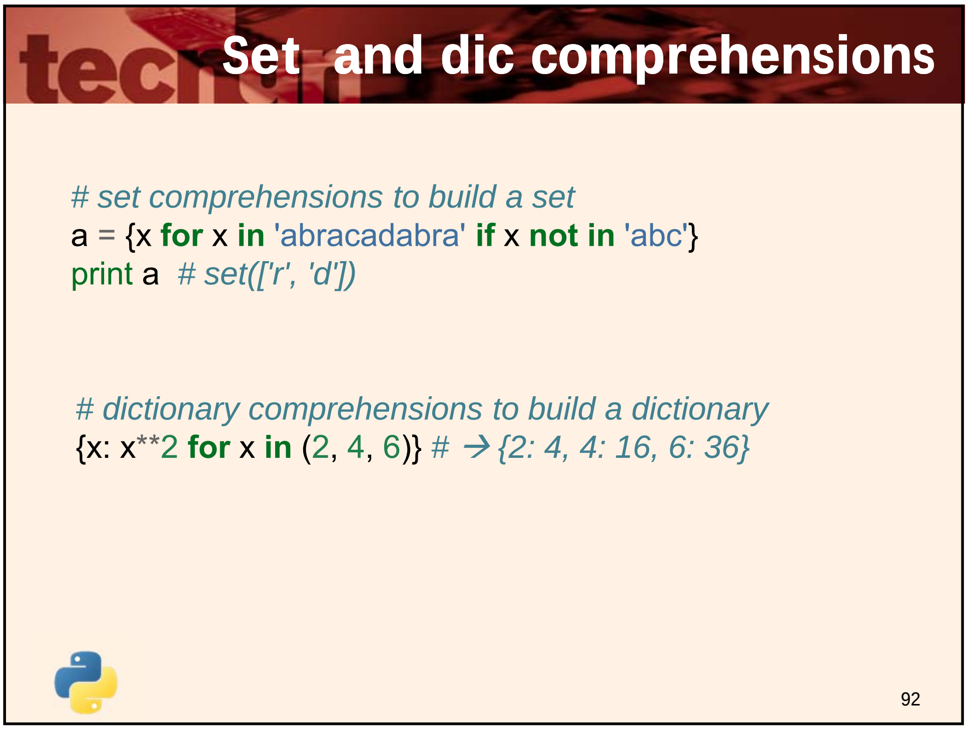
```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12],]
```

- Using a **for** loop and a list comprehension for rows

```
#transpose rows and columns
transposed = []
for i in range(4):
        transposed.append([row[i] for row in matrix])
print transposed
```

- Using a list comprehension without **for** loop

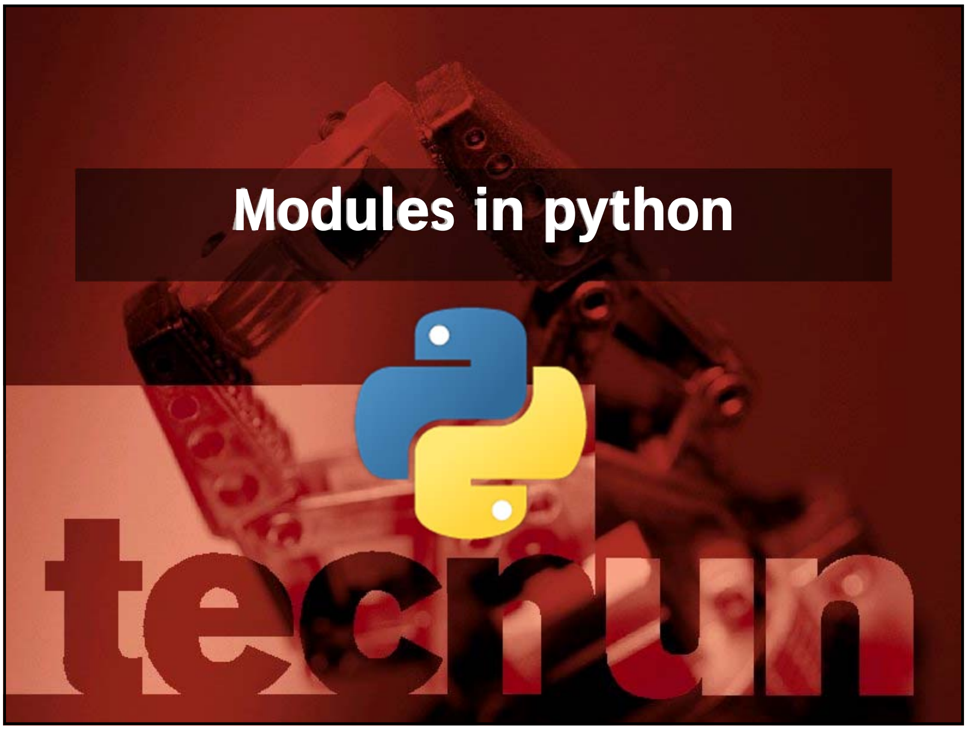
```
#transpose rows and columns
[[row[i] for row in matrix] for i in range(4)]
```



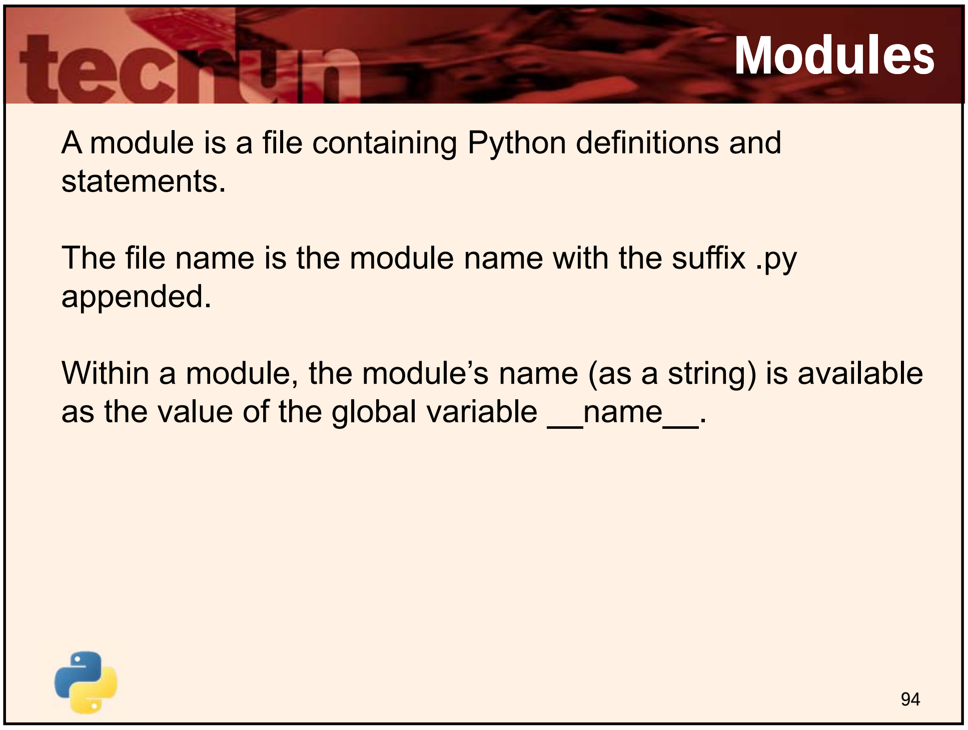
Set and dic comprehensions

```
# set comprehensions to build a set
a = {x for x in 'abracadabra' if x not in 'abc'}
print a  # set(['r', 'd'])
```

```
# dictionary comprehensions to build a dictionary
{x: x**2 for x in (2, 4, 6)} # → {2: 4, 4: 16, 6: 36}
```



Modules in python

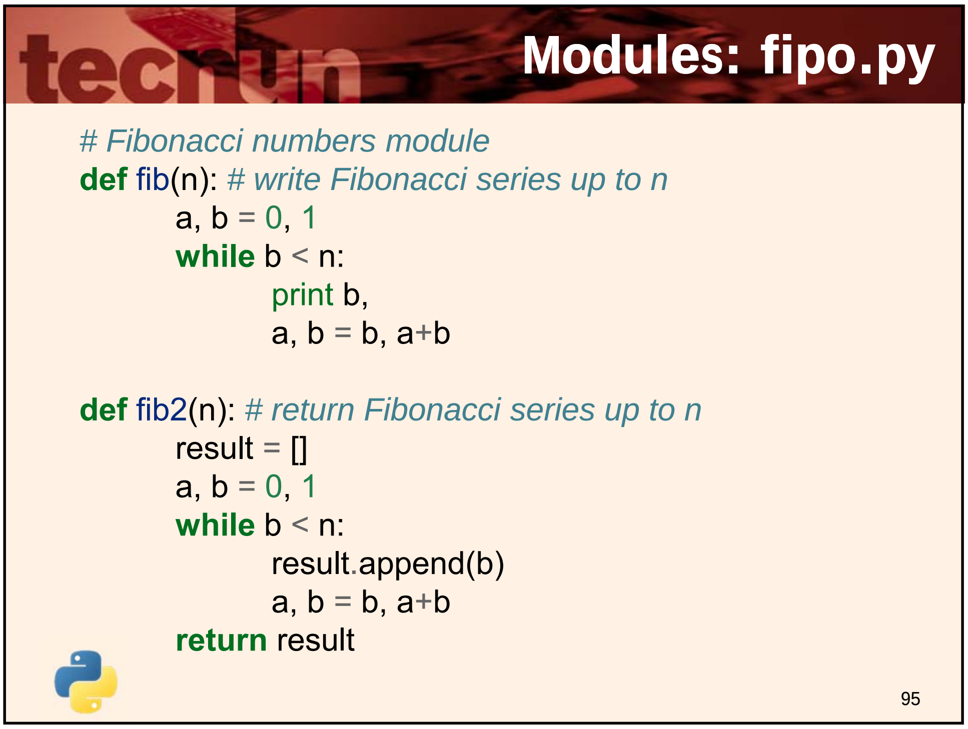


Modules

A module is a file containing Python definitions and statements.

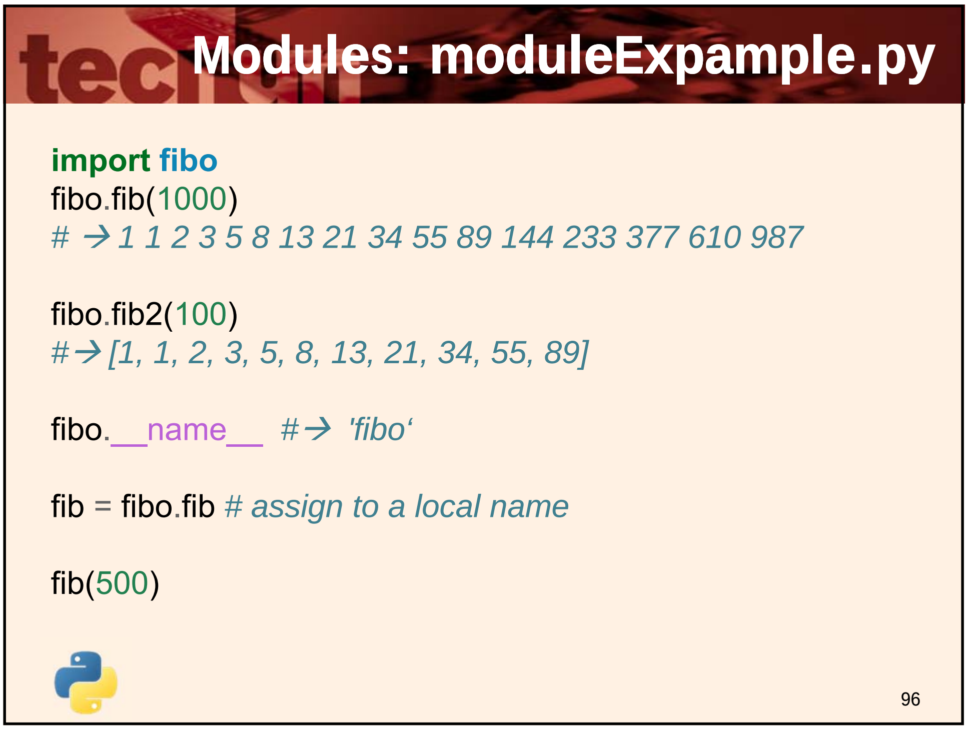
The file name is the module name with the suffix .py appended.

Within a module, the module’s name (as a string) is available as the value of the global variable __name__.



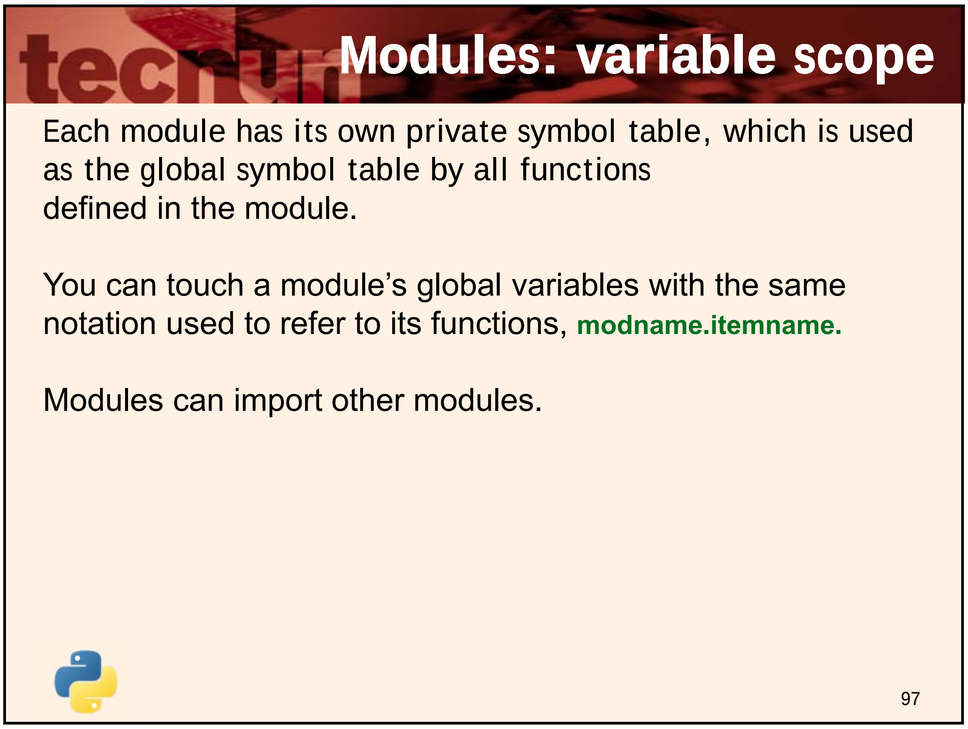
Modules: fipo.py

```
# Fibonacci numbers module
def fib(n): # write Fibonacci series up to n
        a, b = 0, 1
        while b < n:
                print b,
                a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
        result = []
        a, b = 0, 1
        while b < n:
                result.append(b)
                a, b = b, a+b
        return result
```



Modules: moduleExpample.py

```
import fibo
fibo.fib(1000)
# → 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

fibo.fib2(100)
#→ [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

fibo.__name__  #→ 'fibo‘

fib = fibo.fib # assign to a local name

fib(500)
```

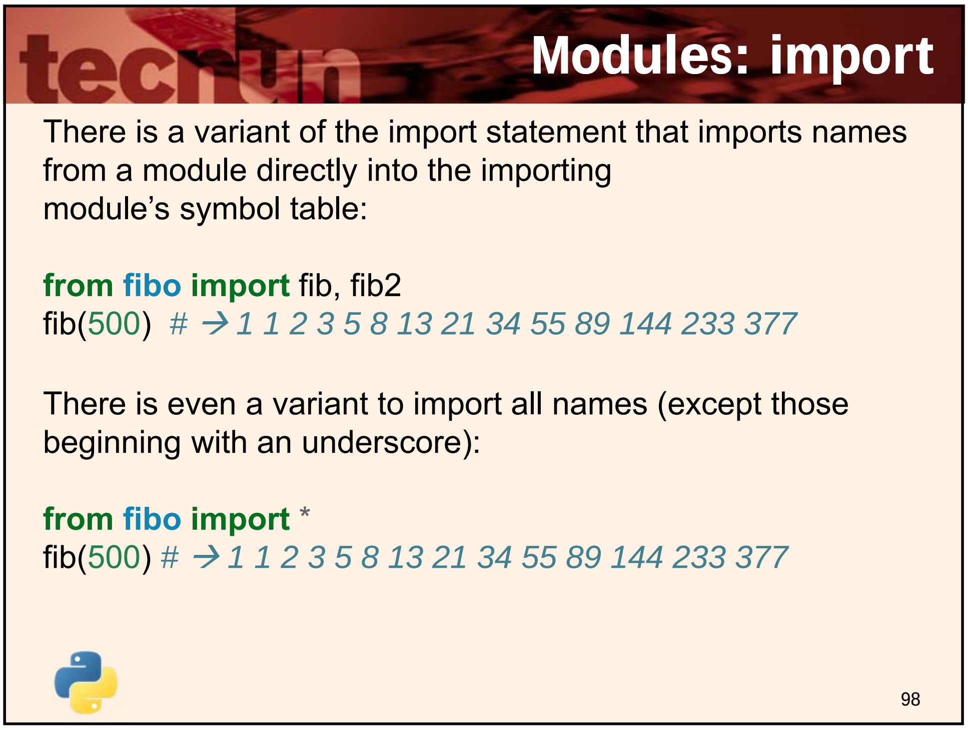


Modules: variable scope

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module.

You can touch a module's global variables with the same notation used to refer to its functions, **modname.itemname.**

Modules can import other modules.



Modules: import

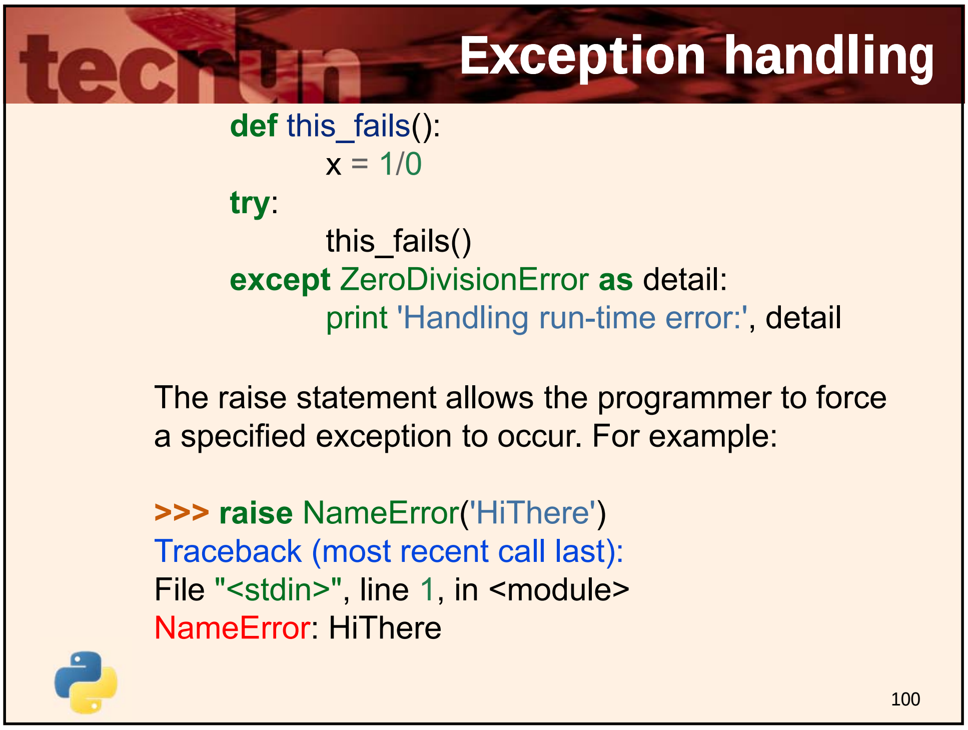
There is a variant of the import statement that imports names from a module directly into the importing module's symbol table:

```
from fibo import fib, fib2
fib(500)  # → 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

There is even a variant to import all names (except those beginning with an underscore):

```
from fibo import *
fib(500) # → 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Exceptions in python

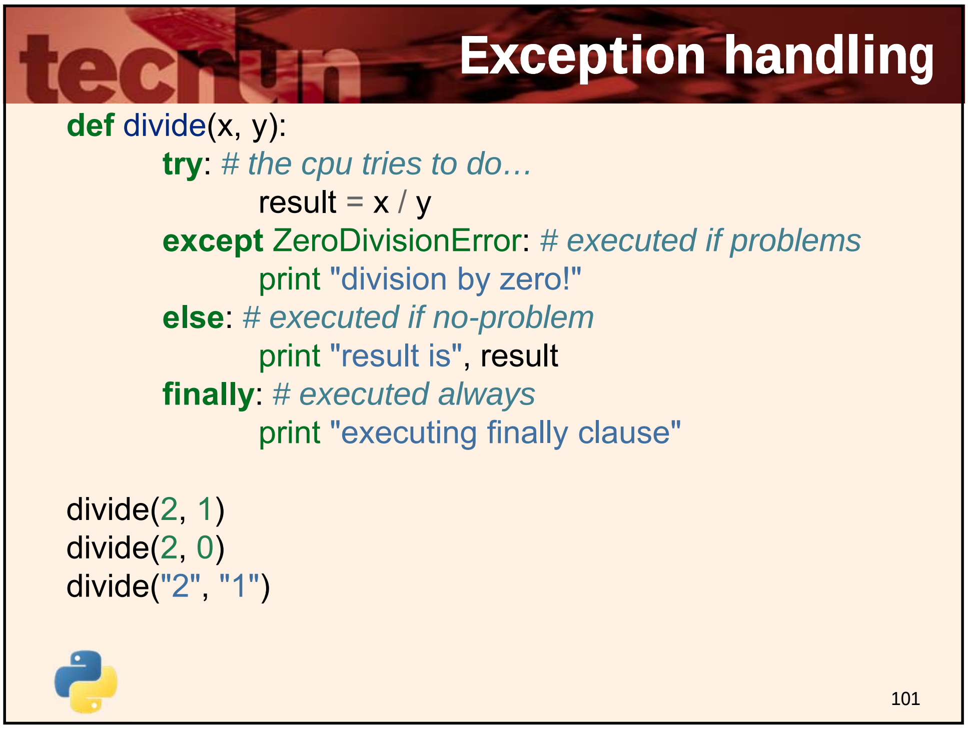


Exception handling

```
def this_fails():
        x = 1/0
try:
        this_fails()
except ZeroDivisionError as detail:
        print 'Handling run-time error:', detail
```

The raise statement allows the programmer to force a specified exception to occur. For example:

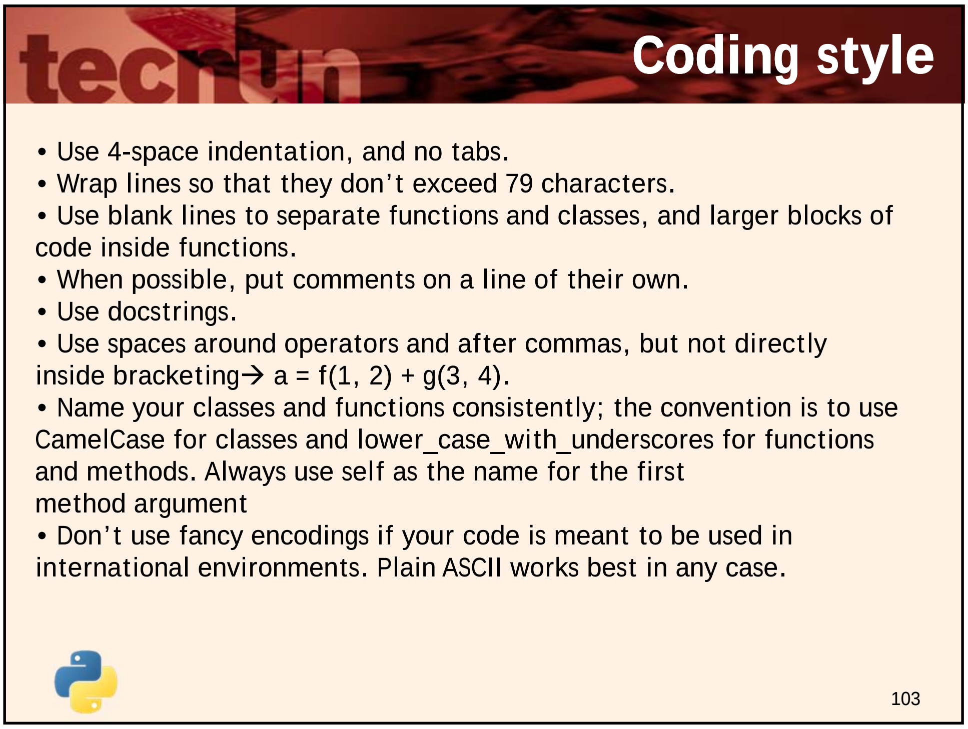
```
>>> raise NameError('HiThere')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: HiThere
```



Exception handling

```
def divide(x, y):
        try: # the cpu tries to do…
                result = x / y
        except ZeroDivisionError: # executed if problems
                print "division by zero!"
        else: # executed if no-problem
                print "result is", result
        finally: # executed always
                print "executing finally clause"

divide(2, 1)
divide(2, 0)
divide("2", "1")
```

Coding style in python



Coding style

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing→ a = f(1, 2) + g(3, 4).
- Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument
- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.