SELECTED TOPICS IN ENGINEERING

INTR. TO PROG. FOR DATA SCIENCE
ENGR 350

Tuesday-Thursday 10:00-12:45
ENG B05
2019 Summer

Dr. Banu Yobaş

Dictionary Details

- Keys must be immutable:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - not lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in arbitrary order
  - again, because of hashing

Dictionaries

- Mutable
- Python dictionary is an unordered collection of items.
- While other compound data types have only value as an element, a dictionary has a

  key: value pair.
- Dictionaries are optimized to retrieve values when the key is known.
- While indexing is used with other container types to access values, dictionary uses keys.
- Key can be used either inside square brackets or with the get() method.

Properties of Dictionary Keys

- Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.
- More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.
- Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Dictionaries

- We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.
- We can remove a particular item in a dictionary by using the method pop().
- This method removes as item with the provided key and returns the value.
- The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary.
- All the items can be removed at once using the clear() method.

What Makes it Special

- The conceptual implementation is that of a hash table, so checks for existence are quite fast.
- That means we can determine if a specific key is present in the dictionary without needing to examine every element (which gets slower as the dictionary gets bigger).
- The Python interpreter can just go to the location key "should be" at (if it's in the dictionary) and see if key is actually there.

Dictionaries

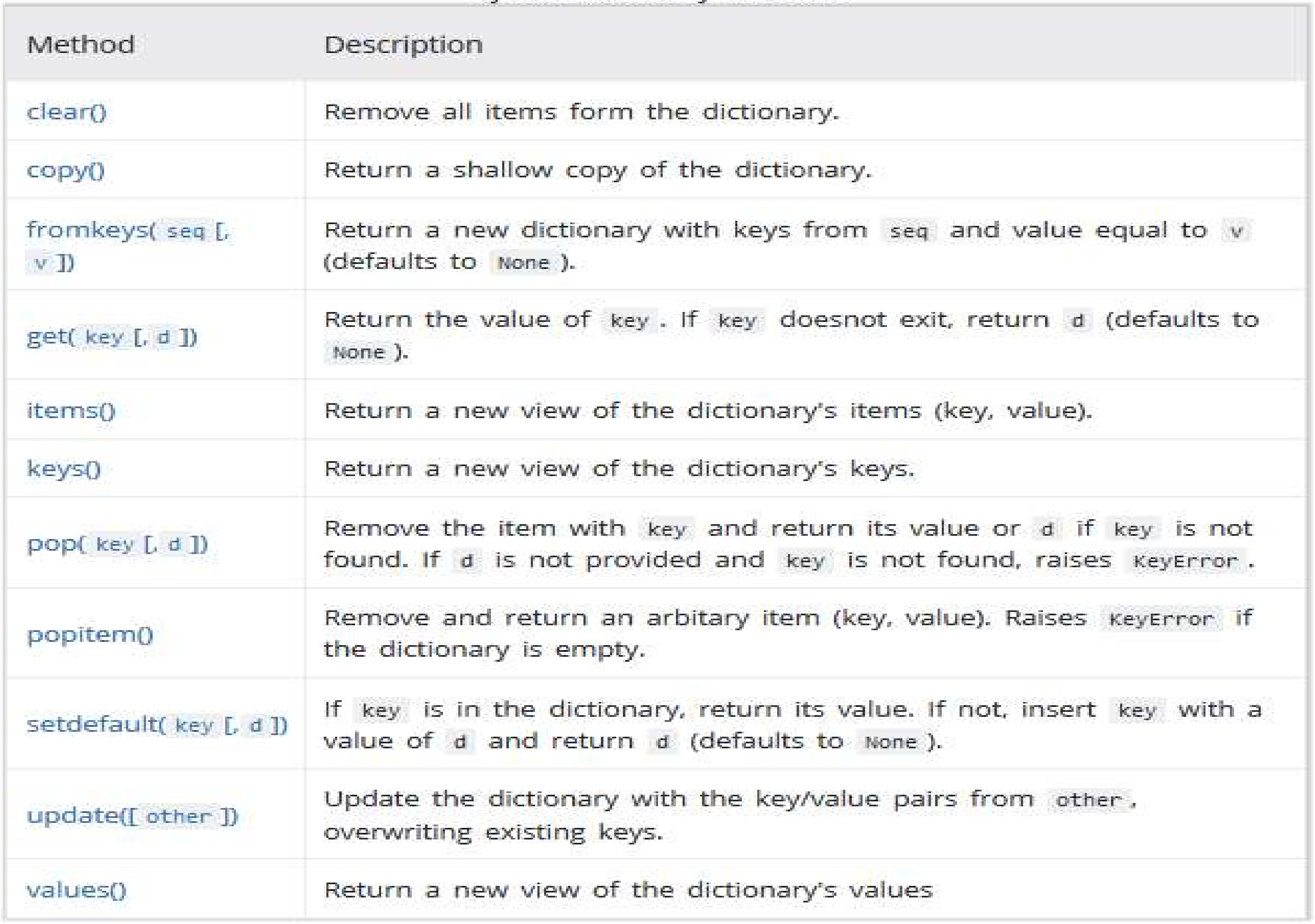
- Hash tables, "associative arrays"
  - d = {"duck": "eend", "water": "water"}
- Lookup:
  - d["duck"] -> "eend"
  - d["back"] # raises KeyError exception
- Delete, insert, overwrite:
  - del d["water"] # {"duck": "eend", "back": "rug"}
  - d["back"] = "rug" # {"duck": "eend", "back": "rug"}
  - d["duck"] = "duik" # {"duck": "duik", "back": "rug"}

More Dictionary Ops

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1,
    42:"yes", "flag": ["red","white","blue"]}

Dictionaries

Python Dictionary Methods



Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(`seq` [, `v`])	Return a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(`key` [, `d`])	Return the value of `key` . If `key` doesnot exit, return `d` (defaults to `None`).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(`key` [, `d`])	Remove the item with `key` and return its value or `d` if `key` is not found. If `d` is not provided and `key` is not found, raises `KeyError` .
popitem()	Remove and return an arbitary item (key, value). Raises `KeyError` if the dictionary is empty.
setdefault(`key` [, `d`])	If `key` is in the dictionary, return its value. If not, insert `key` with a value of `d` and return `d` (defaults to `None`).
update([`other`])	Update the dictionary with the key/value pairs from `other` , overwriting existing keys.
values()	Return a new view of the dictionary's values

Example

- You are given a list of N numbers. Create a single list comprehension in Python to create a new list that contains only those values which have even numbers from elements of the list at even indices.
- For instance if list[4] has an even value the it has be included in the new output list because it has an even index but if list[5] has an even value it should not be included in the list because it is not at an even index.

Dictionary Comprehension

- Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.
- Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces {}.

```
squares = {x: x*x for x in range(6)}
        # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
print(squares)
```

Dictionary Comprehension

```
squares = {x: x*x for x
  in range(6)}

# Output: {0: 0, 1: 1, 2: 4, 3: 9,
  4: 16, 5: 25}
print(squares)
```

```
squares = {}
for x in range(6):
   squares[x] = x*x
```

Comprehension

Using loop

Dictionary Comprehension

- A dictionary comprehension can optionally contain more for or if statements.
- An optional if statement can filter out items to form the new dictionary.

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}

# Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
print(odd_squares)
```

Dictionary Operations

- test if a key is in a dictionary or not using the keyword in
- Notice that membership test is for keys only, not for values.
- Using a for loop we can iterate through each key in a dictionary.

- Use dictionary whenever a mapping from a key to a value is required.

How Not to Use It

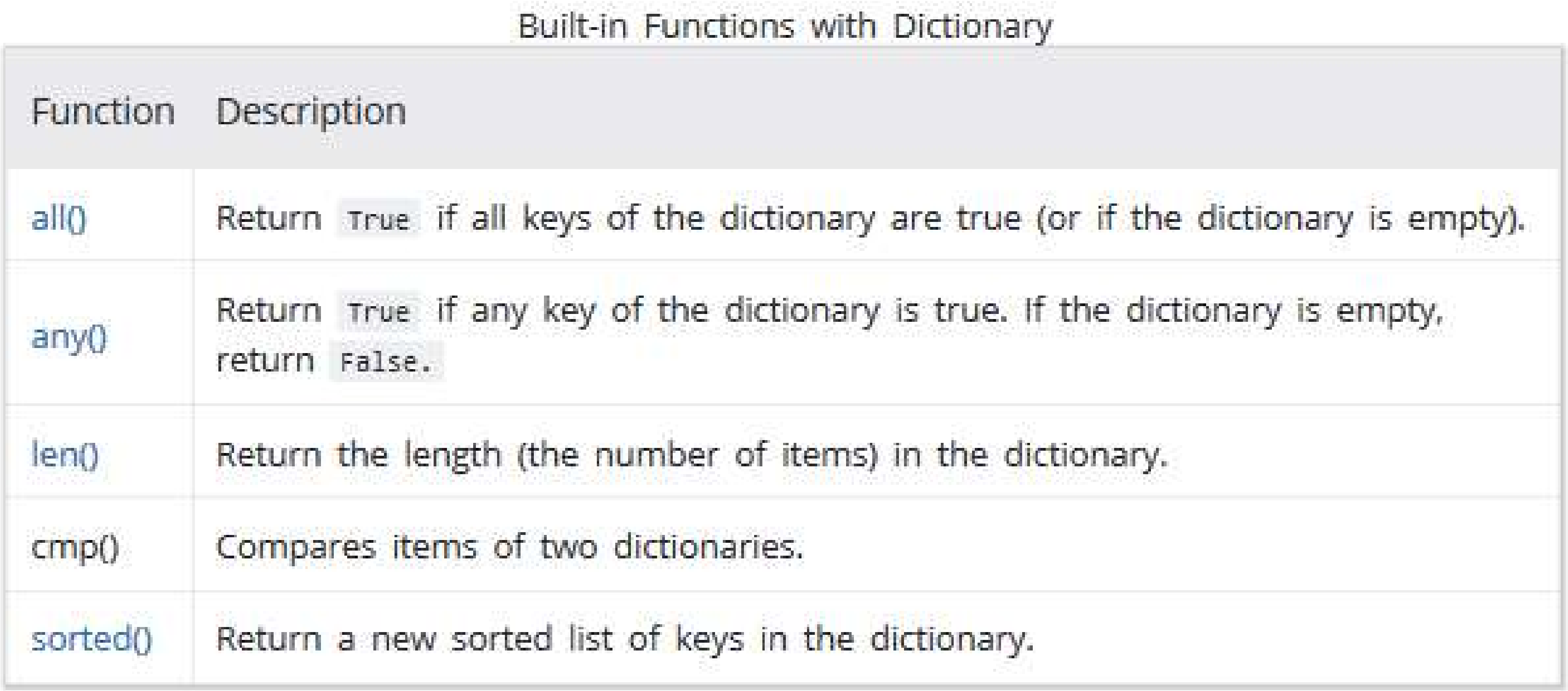
- Remember, the great thing about dictionaries is we can find a value instantly, without needing to search through the whole dictionary manually, using the form

value = my_dict['key'] or

value = my_dict.get('key', None).

- If you're searching for a value in a dictionary and you use a for loop, you're doing it wrong. Stop, go back, and read the previous statement.

Built-in Functions with Dictionary



Built-in Functions with Dictionary

Function	Description
all()	Return `True` if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

Which one to use?

- list keeps order,
- list and set just contain values
- list doesn't
- list does not
- in a list it takes time proportional to the list's length in the average and worst cases.

Lists

- dict and set don't
- dict associates with each key a value
- set requires items to be hashable,
- set forbids duplicates,
- checking for membership of a value in a set (or dict, for keys) is blazingly fast (taking about a constant, short time),

Set & Dictionary

NONE – NULL

- None is a special constant in Python. It is a null value.
- None is not the same as False.
- None is not 0.
- None is not an empty string.
- Comparing None to anything other than None will always return False .
- None is the only null value. It has its own datatype (NoneType).
- You can assign None to any variable, but you can not create other NoneType objects.
- All variables whose value is None are equal to each other.

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

Assignment

- Python evaluates the right side before it assigns a value to a variable
- Lets multiple assignments

```
>>> x, y, z = 1, 'a' ,True
>>> x,y,z
(1, 'a', True)
>>> x
1
>>> y
'a'
>>> z
True
```

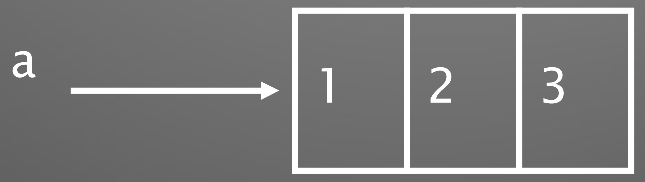
Reference Semantics

- Assignment manipulates references
  - x = y does not make a copy of y
  - x = y makes x reference the object y references
- Very useful; but beware!
- Example:

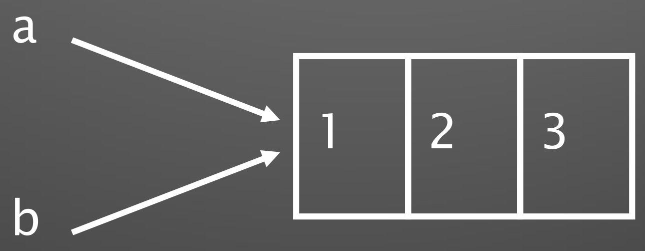
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Changing a Shared List

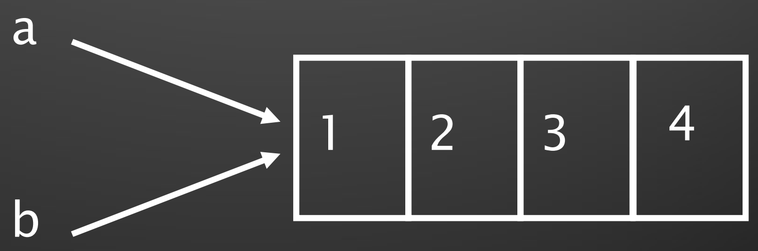
```
a = [1, 2, 3]
```



```
b = a
```



```
a.append(4)
```

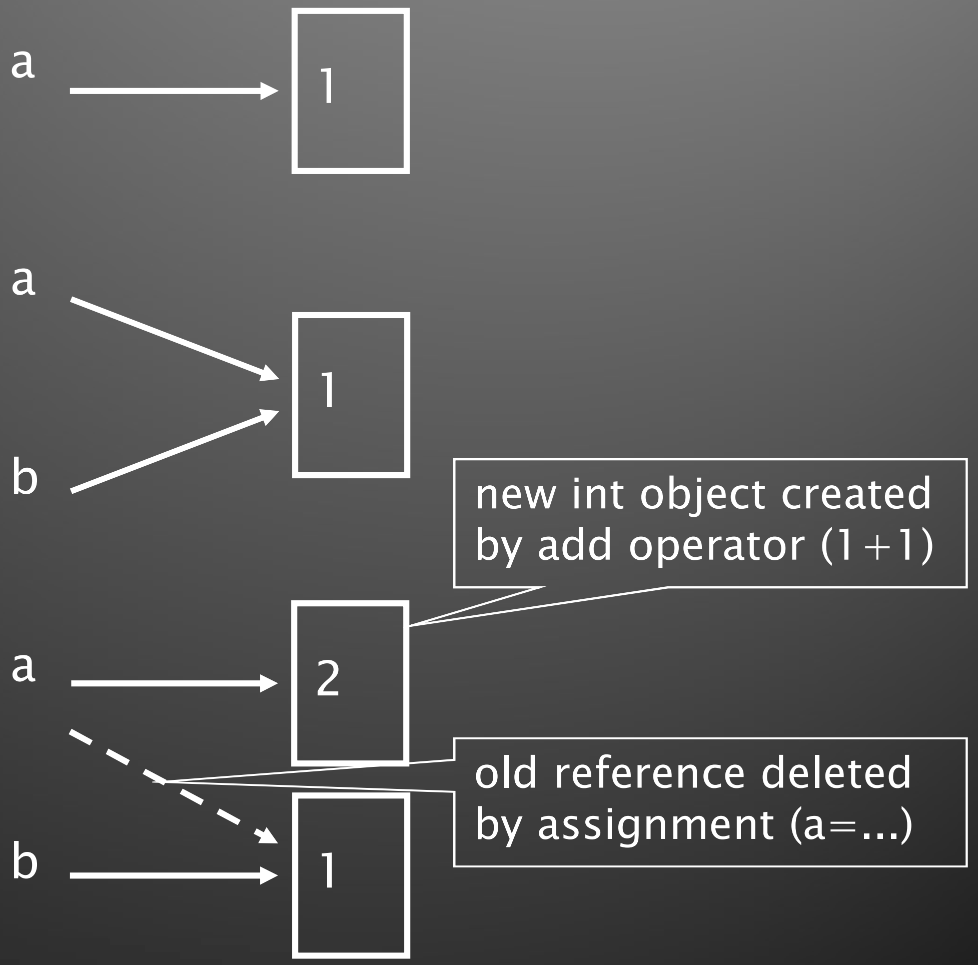


Changing an Integer

```
a = 1

b = a

a = a+1
```



a
1
a
1
b
new int object created by add operator (1+1)
a
2
old reference deleted by assignment (a=...)
b
1

None – List, String

- Most list methods modify the argument and return None.
- This is the opposite of the string methods, which return a new string and leave the original alone.

Iteration types

- Definite loops
- Indefinite/conditional loops
- Infinite loops (Opps!)

Definite loops

- Number of iterations <u>can</u> be determined
- For loops
- The variable after the keyword for is called the *loop index.*

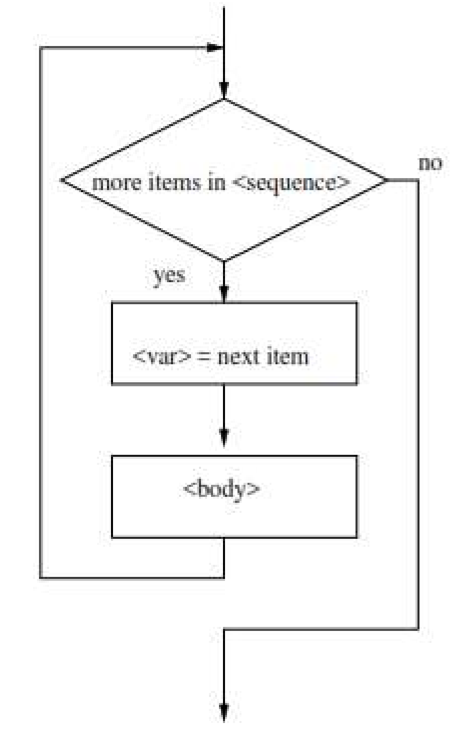


Figure 2.3: Flowchart of a `for` loop.

Definite Loops/iterations

```
for <var> in <sequence>:
    <body>
```

- The body of the loop can be any sequence of Python statements. The start and end of the body is indicated by its indentation under the loop heading (the for <var> in <sequence>: part).

```
for i in [0,1,2,3]:  #loop index is i
    print i
```

Indefinite loops

- Number of iterations <u>can NOT</u> be determined before runtime
- While loops
- The boolean variable used as the condition is called a Boolean flag

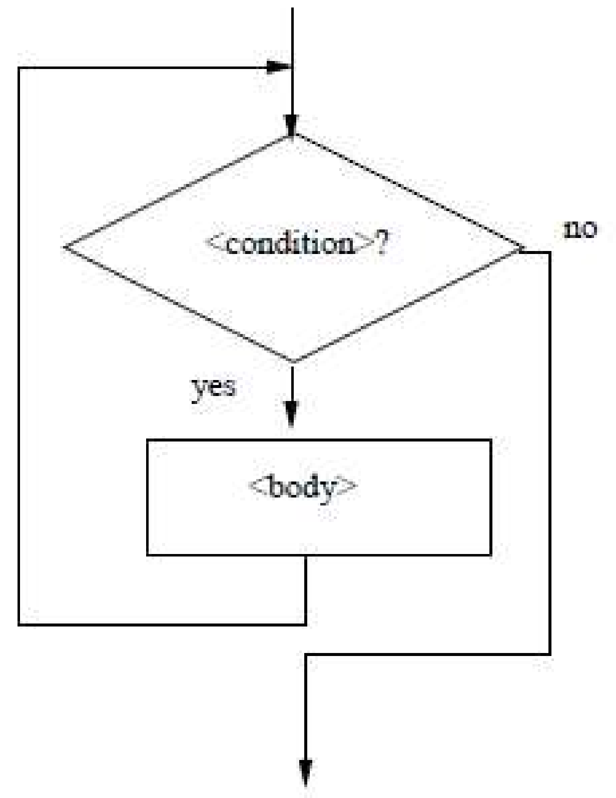


Figure 8.1: Flowchart of a while loop.

Indefinite loops/iterations

```
while <condition>:
    <body>
```

Flow of execution for a while statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

Common Loop Patterns

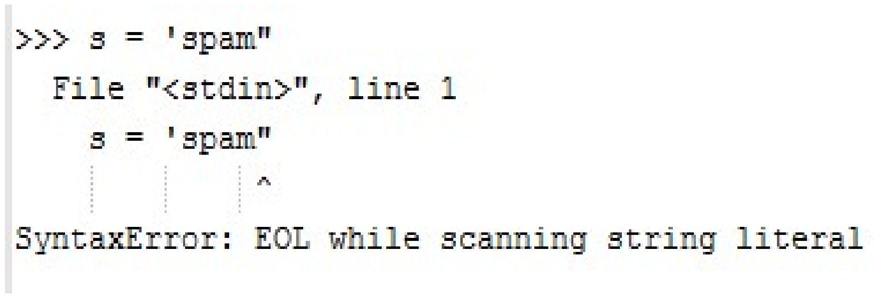
- Interactive loops
  - The idea behind an interactive loop is that it allows the user to repeat certain portions of a program on demand.
- Sentinel loops
  - sentinel loop continues to process data until reaching a special value that signals the end. The special value is called the *sentinel*. Any value may be chosen for the sentinel. The only restriction is that it be distinguishable from actual data values. The sentinel <u>is not processed</u> as part of the data.
- end-of-file loops
  - This file-oriented approach is typically used for data processing applications.

Loops

- Continue
- Break
- Nested Loops

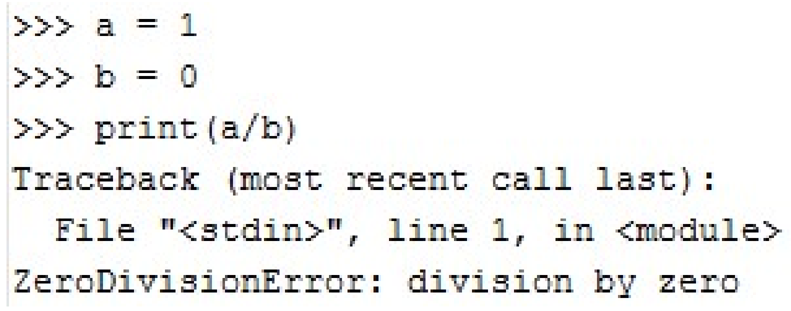
Three kinds of errors

- Syntax error:



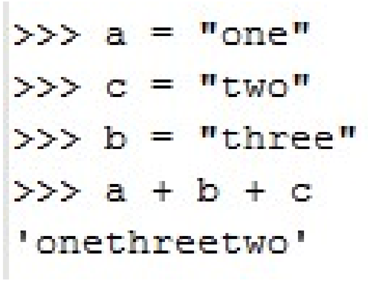
```
>>> s = 'spam"
  File "<stdin>", line 1
    s = 'spam"
             ^
SyntaxError: EOL while scanning string literal
```

- Runtime error:



```
>>> a = 1
>>> b = 0
>>> print(a/b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- Semantic error:



```
>>> a = "one"
>>> c = "two"
>>> b = "three"
>>> a + b + c
'onethreetwo'
```

You program runs to completion,
but produces the wrong output

Errors and Expressions

- Syntactic error invalid character etc

  Error message
- Semantic error when the logic is wrong

  NO error message

Runtime errors:

- NameError:
- TypeError:
- KeyError:
- …

Modularization

- Breaking down a large main program into modules
- Statements taken out of a main program and put into a module have been encapsulated
- Main program shorter and easier to understand
- Reasons
  - Abstraction
  - Allows multiple programmers to work on a problem
  - Reuse your work more easily

Reusability & Reliability

Reusability

- Feature of modular programs
- Allows individual modules to be used in a variety of applications
- Many real-world examples of reusability

Reliability : Assures that a module has been tested and proven to function correctly

Functions, Procedures

```
def name(arg1, arg2, ...):
    """documentation"""          # optional doc
  string
    statements

return                           # from procedure
return expression                # from function

Pass
```

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

Function return values

- Void functions might display something on the screen or have some other effect, but they don’t have a return value.
- If you assign the result to a variable, you get a special value called None

```
result= a_void_func(a,b)
print(result)
line=[‘a’, ‘x’ ‘dag’ ‘cb’ ‘bac’]
print(line.sort())
line
```

Function()

- Sometimes a function needs to return more than one value. This can be done by simply listing more than one expression in the return statement.
- As a silly example

```
def sumDiff(x,y):
    sum = x + y
    diff = x - y
    return sum, diff
```

- This return hands back two values. When calling this function, we would place it in a simultaneous assignment.

```
num1, num2 = input("Please enter two numbers (num1,
  num2) ")
s, d = sumDiff(num1, num2)
print "The sum is", s, "and the difference is", d
```

Modules

- math
- string
- random
- os
- system