Programming for Data Science

(BUAN 6340)

Lecture Python Basics

Yingjie Zhang
University of Texas at Dallas
yingjie.zhang@utdallas.edu
Fall 2019

Agenda

- Python beginner
- Data types
- Data Structures
- Functions
- Control Flows

Indentation

- Python uses indentation for blocks, instead of curly braces. Both tabs and spaces are supported, but the standard indentation requires standard Python code to use four spaces.

```
x = 1
if x == 1:
    # indented four spaces
    print("x is 1.")
```

Comments

Single line comments start with a '#'

"""

Multiline comments can be written between three ''s and are often used as function and module comments

"""

Print

- Output some contents to the screen (or some files)
  - Use comma-separated arguments or concatenate strings.
  - Each argument will be evaluated and converted to a string for output.

```
>>> print(555, 867, 5309)
555 867 5309
>>> print(555, 867, 5309, sep="-")
555-867-5309
```

Data Types

Data Types

- Python contains some basic data types, including number (int, float, complex), string, list, dictionary, etc.
- All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.
- Prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
    - Example: 2 + “2” will report error

Numbers

- Numeric
  - **int**: equivalent to C's long int in 2.x but unlimited in 3.x.
  - **float**: equivalent to C's doubles.
  - **complex**: complex numbers.
- Mixed arithmetic is supported, with the "narrower" type widened to that of the other, , where integer is narrower than floating point, which is narrower than complex.
  - 2*3.1

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

Boolean

- Bool is a subtype of int, where True == 1 and False == 0.

```
True              # => True
False             # => False

not True          # => False
True and False    # => False
True or False     # => True (short-circuits)

1 == 1            # => True
2 * 3 == 5        # => False
1 != 1            # => False
2 * 3 != 5        # => True

1 < 10            # => True
2 >= 0            # => True
1 < 2 < 3         # => True  (1 < 2 and 2 < 3)
```

Strings

- Created by simply enclosing characters in either single- or double-quotes.
  - Matched quotes will be considered starting/end of the string
  - Supports a number of escape sequences such as '\t', '\n', etc.
- Strings are immutable (read-only).
  - Two string literals beside one another are automatically concatenated together.
- Conversion between number str(), int(), float()

String -- indexing

"\"

\n -> newline

\t -> tab

\\ -> backslash

...

Note that Windows uses backslash for directories!

filename = "M:\nickel_project\reactive.smi" # DANGER!

filename = "M:\\nickel_project\\reactive.smi" # Better!

filename = "M:/nickel_project/reactive.smi" # Usually works

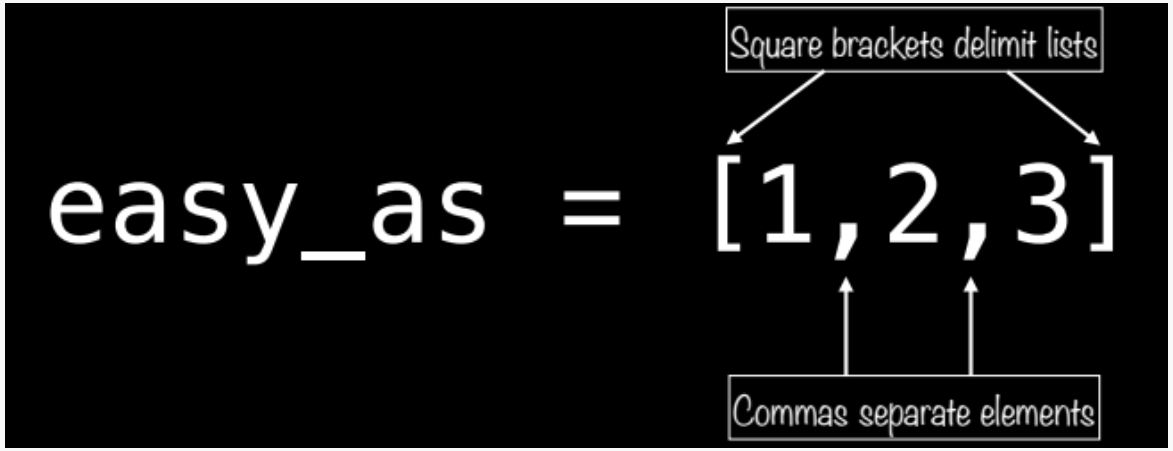
Data Structures

- Lists
- Dictionaries
- Tuples
- Set

LISTS

Lists

- Lists are an list of elements
  - The elements can be different data types, such as string, number, or list (lists are nestable)
  - The order of the elements matter
- Lists can be initialized by a bracket structure
  - It can be empty
- Lists are mutable (editable).



Creation and Access

- To create a list in Python, we can use bracket notation to either create an empty list or an initialized list.

  the list() function

```
mylst1=[0]
mylst2=[2.3, 4.5]
mylst3=[5, "Hello", "there", 9.8]
mylst4=[]
mylst5= [1, 2, ['item1', 'item2'], ['item3', 'item4']]
```

- Use [ ] to Access Items in the List
  - [0] is the first item, [1] is the second item, …
  - Out of range values cause an error (raise an exception)
  - Negative values go backwards from the last element, -1 is the last element

```
print(mylst2[1], mylst3[2], mylst5[3])
print(mylst2[-1], mylst3[-2], mylst5[-3])
```

Slicing

- The length of the list is accessible through `len(mylist)`.
- Slicing is an extended version of the indexing operator and can be used to grab sublists.

```
mylist[n1:n2]          # items from n1 to n2-1
mylist[n1:]            # items from n1 to end
mylist[:n1]            # items from beginning to n1-1
mylist[:]              # a copy of the whole list
```

- You may also provide a step argument with any of the slicing constructions above.
- The start or end arguments may be negative numbers, indicating a count from the end.

```
mylist[n1:n2:step] # n1 to n2-1, by step
```

Useful Methods

```
>>> mylst = ["spam", "egg", "toast"]
>>> "egg" in mylst # whether an element is in a list
True
>>> len(mylst)  # get the length of a list
3
>>> mynewlst = ["coffee", "tea"]
>>> mylst + mynewlst # concatenate two lists together
['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mylst.append("apple") # append an element to the end of the list
>>> mylst
['spam', 'egg', 'toast', 'apple']

>>> mylst.index('apple') # find the first index of an item
3
```

Useful Methods

```
>>> del mylst[0] # remove an element
>>> mylst
['egg', 'toast', 'apple']
>>> mylst.sort() # sort by default order
>>> mylst
['apple', ‘egg', ‘toast']
>>> mylst.reverse() # reverse the elements in a list

>>> mylst
['toast', 'egg', 'apple']
>>> mylst.insert(0, "spam") #  insert an element at some specified position.
                               (Slower than .append())
>>> mylst
['spam', 'egg', 'toast', 'apple']


>>> mylst.count('apple') # Count the number of occurrences

1
```

When to use Lists

- Heterogeneous elements.
- Order the elements.
- Modify the collection.
- Need a stack or a queue.
- Elements are not necessarily unique.

Common Sequence Operations

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]	ith 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.
s.index(x)	Index of the first occurrence of x in s.
s.count(x)	Total number of occurrences of x in s.

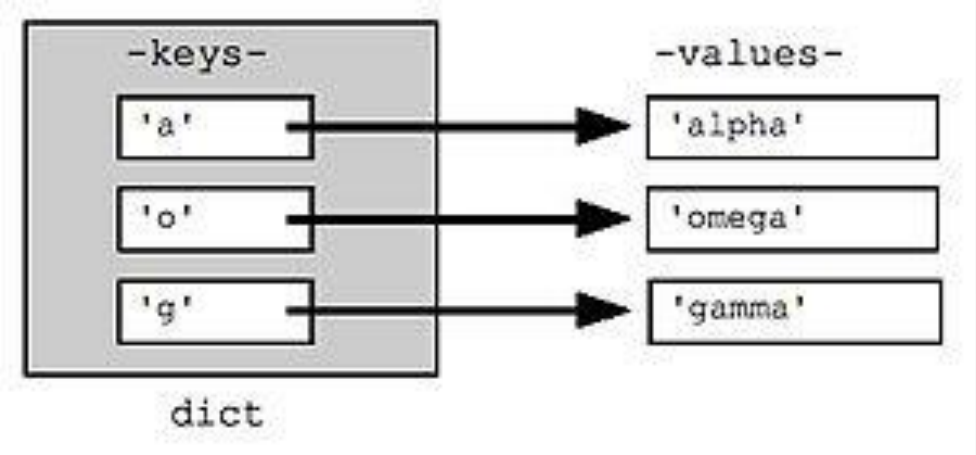
Common Sequence Operations

Operation	Result
s[i] = x	Item i of s is replaced by x.
s[i:j] = t	Slice of s from i to j is replaced by the contents of t.
del s[i:j]	Same as s[i:j] = [].
s[i:j:k] = t	The elements of s[i:j:k] are replaced by those of
del s[i:j:k]	Removes the elements of s[i:j:k] from the list.
s.append(x)	Same as s[len(s):len(s)] = [x].
s.extend(x)	Same as s[len(s):len(s)] = x.
s.count(x)	Return number of i's for which s[i] == x.
s.index(x[, i[, j]])	Return smallest k such that s[k] == x and i <= k < j.
s.insert(i, x)	Same as s[i:i] = [x].
s.pop(i)	Same as x = s[i]; del s[i]; return x.
s.remove(x)	Same as del s[s.index(x)].
s.reverse()	Reverses the items of s in place.
s.sort([cmp[, key[, reverse]]])	Sort the items of s in place.

DICTIONARIES

Dictionaries

- Dictionaries are lookup tables.
  - Map a “key” to a “value”.
  - Duplicate keys are not allowed
  - Duplicate values are fine
- You can initialize a dictionary by specifying each key:value pair within the curly braces.



Create a Dictionary

```
>>> empty = {}
>>> type(empty)
dict

>>> empty == dict()
True

>>> a = dict(one = 1,two = 2,three = 3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> a == b
True
```

Access and Mutate

```
>>> d = {'one': 1, 'two': 2, 'three': 3}

# Get
>>> d['one']
1
>>> d['five']           # raises KeyError

# Set
>>> d['two'] = 22       # modify an existing key
>>> d['four'] = 4       # add a new key



# use get() to avoid the KeyError
>>> d = {'BUAN': [106,107,110], 'MIS': [51,113]}
>>> d.get('BUAN')
[106,107,110]
>>> d.get('ACOOUNT')
None
```

Common Dict Operations

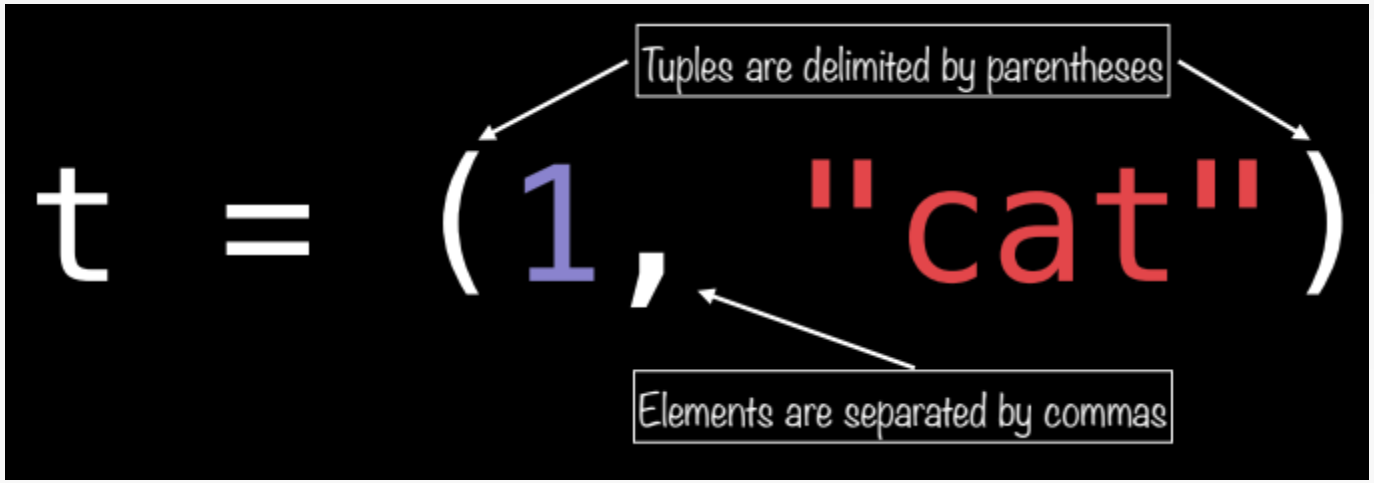
- d.keys()
- d.values()
- d.items()
- len(d)
- key in d
- value in d.values()
- d.copy()
- d.clear()

TUPLES

Tuples

Different from list:

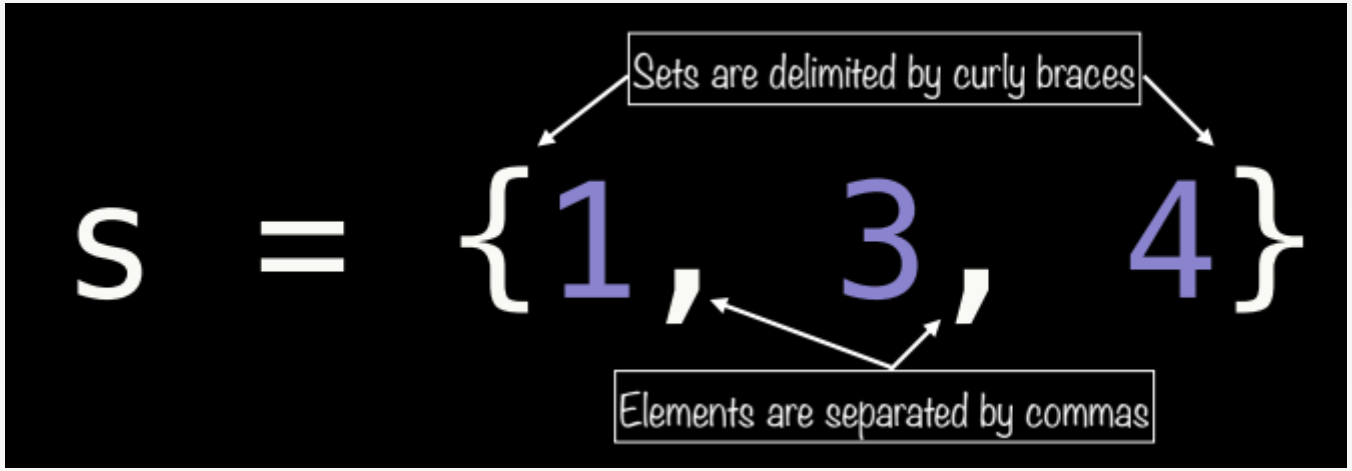
- typed with parentheses
- immutable



SETS

Sets

- Set is an unordered collection of distinct elements




```
>>> empty_set = set()
>>> s = set([1,2,1,4,3])
{1,3,4,2}
```

Common Set Operations

```
a = set("mississippi")  # {'i', 'm', 'p', 's'}

a.add('r')
a.remove('m')  # raises KeyError if 'm' is not present
a.discard('x')  # same as remove, except no error

a.pop()  # => 's' (or 'i' or 'p')
a.clear()
len(a)  # => 0
```

Common Set Operations

```
a = set("abracadabra")  # {'a', 'r', 'b', 'c', 'd'}
b = set("alacazam")     # {'a', 'm', 'c', 'l', 'z'}

# Set difference
a - b  # => {'r', 'd', 'b'}

# Union
a | b  # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

# Intersection
a & b  # => {'a', 'c'}

# Symmetric Difference
a ^ b  # => {'r', 'd', 'b', 'm', 'z', 'l'}
```

Summary

- Lists [items]
- Dictionaries {key: value}
- Tuples (frozen, sequence)
- Sets {unique, values}

Functions

Basic Elements

```
def fn_name(param1, param2):
    value = do_something()
    return value
```

- A function is created with the def keyword.
  - The statements in the block of the function must be indented.
  - The arguments are after the function name in round brackets.
  - The return keyword, if used can specify a list of values to be returned.
- Function calls needs to be put after the function definitions.

Return

- All functions return some value
  - Even if that value is None
- No return statement or just return implicitly returns None
- Returning multiple values
  - You can use a tuple! In some cases, use a named tuple
  - return value1, value2, value3

Functions

- Default argument values
  - We can provide a default value for any number of arguments in a function.
  - Allows functions to be called with a variable number of arguments.
  - Arguments with default values must appear at the end of the arguments list!
- The default arguments are initialized once when the function is defined/imported, not when the function is called.

```
def print_greeting(name, year="2017"):
    print("Hello,", name, year,"!\n")

print_greeting("Ben")
print_greeting("Ben","2016")
```

Functions

- *keyword argument*
  - By using keyword arguments, we can explicitly tell Python to which formal parameter the argument should be bound. Keyword arguments are always of the form *kwarg = value*.
  - If keyword arguments are used they must follow any *positional arguments*, although the relative order of keyword arguments is unimportant.

```
def print_greeting(name="Ben", year="2017"):
    print("Hello,", name, year,"!\n“)

print_greeting(name="Ben",year="2012")
print_greeting(year="2012",name="Ben")
```

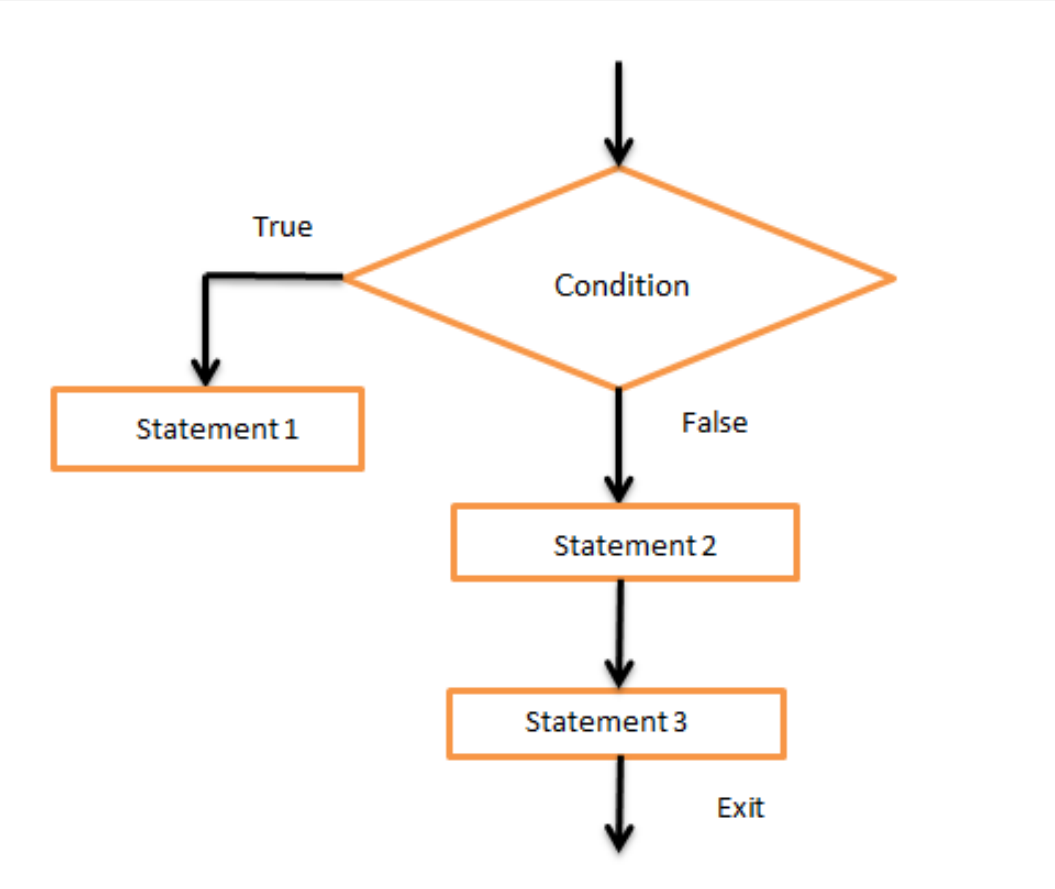
Local and Global Scope

- Function execution introduces a new local symbol table (scope)
    - Local variables cannot be used in the global scope or other local scopes
    - Global variables can be read from a local scope
    - Avoid using local variables that have the same name as a global variable or another local variable

Control Flows

Overview

- Controls the process of the program
  - Given specified conditions are met
  - Conditions could be true or false
    - Things that are False
    - Things that are True



if … elif … else

```
If [condition]:
        [operations]
elif [condition]:
        [operations]
else:
        [operations]
```

- The corresponding operations are executed if the condition is true.
- elif and else statements are optional.
- If statements can be nested

If vs. elif

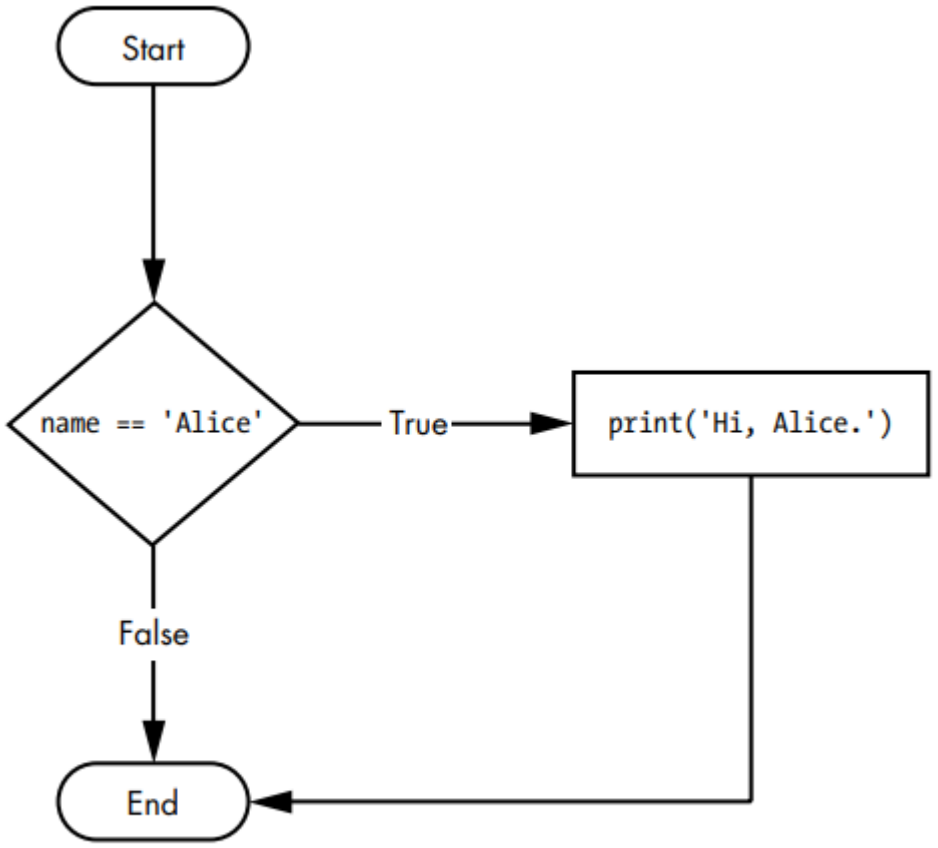


Figure 2-3: The flowchart for an if statement

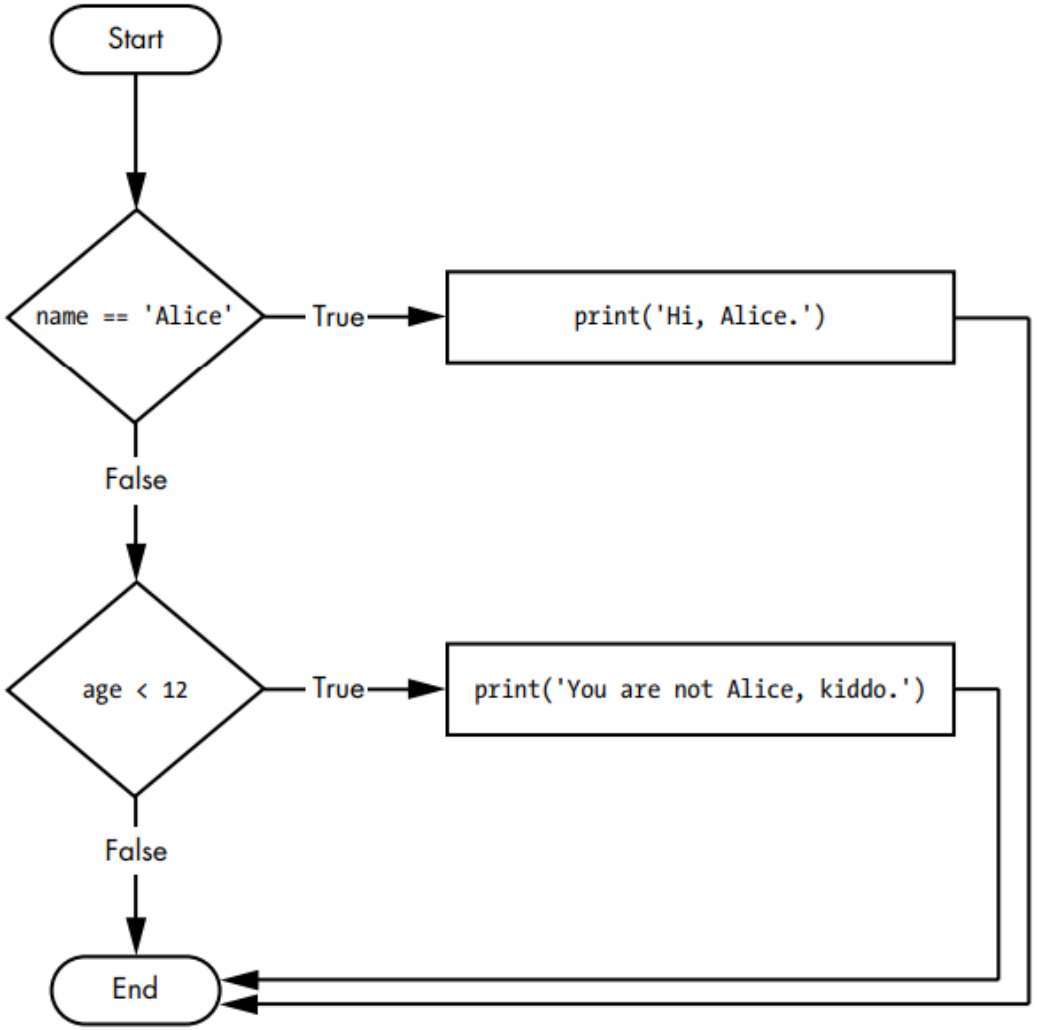


Figure 2-5: The flowchart for an elif statement

while

while [condition]:

[operations]

- The operations are iterated as long as the condition is true.

```
i = 1
while i < 4:
    print i
    i = i + 1
flag = True
while flag and i < 8:
    print flag, i
    i = i + 1
```

while

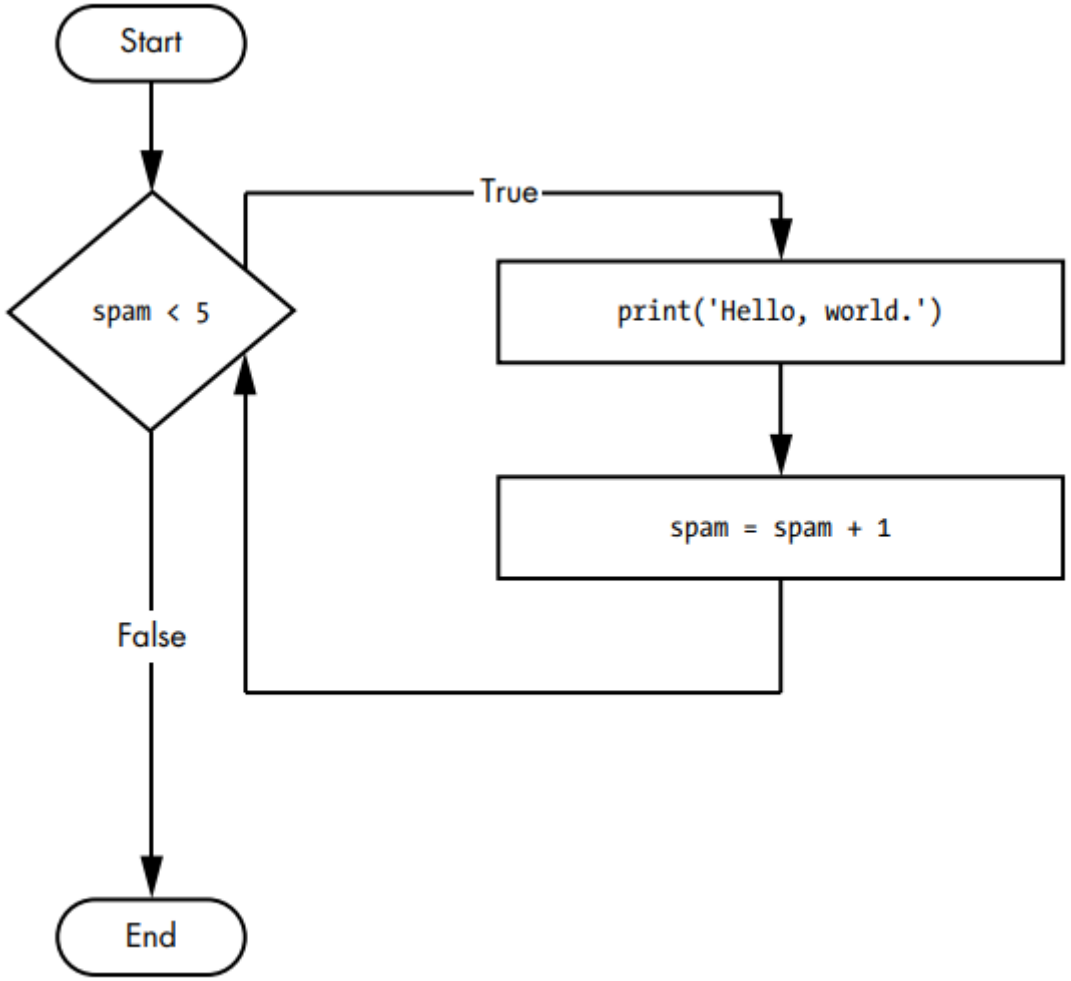


Figure 2-10: The flowchart for the while statement code

for

for [var] in [sequence]:

    [operations]

- The items in the sequence is assigned into the variable var one by one.
- In each round, the operations are executed until the entire sequence is exhausted.
  - var is not necessary to be used in the operations
  - var can be compound types
- Python can create a range of integers, typically used in for loops
  - range([start,] stop[, step]) -> list of integers
  - When step is given, it specifies the increment (or decrement).

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

Control Flow Tools

- There are four statements provided for manipulating loop structures.
  - break – terminates the current loop.
  - continue – immediately begin the next iteration of the loop.
  - pass – do nothing. Use when a statement is required syntactically.
  - else – represents a set of statements that should execute when a loop terminates without any 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')
```

Questions?