EASC2410 Lecture 3

Python Basics: Dictionaries, Program Control

Dr. Binzheng Zhang
Department of Earth Sciences

Review of Lecture 2

In Lecture 2, we learned:

- **How to create variables with meaningful (descriptive) names**
- **How to assign values to variables (use the “=” sign)**
- **What are the basic data types in Python (int, str, float, bool, etc.)**
- **What operators can be used to different data types**
- **What are the basic data structures in Python (sets, tuples, lists)**

In Lecture 3, you will learn:

- **One more interesting data structure in Python: Dictionaries**
- **Program control using for, if, while loops with booleans**

Data structure: Dictionaries in Python

Dictionaries are denoted by curly brackets { } (similar to sets). Dictionaries are somewhat like lists, but denoted by *alphanumeric* keys rather than integer indices. Here's what a dictionary data structure looks like:

```
solar_planets = {'Mercury':2440.0, 'Venus':6052.1, 'Earth':6371.5, 'Mars':3389.2, 'Jupiter':69911.0, 'Saturn':58232.2}
print(type(solar_planets), solar_planets)
```

```
<class 'dict'> {'Mercury': 2440.0, 'Venus': 6052.1, 'Earth': 6371.5, 'Mars': 3389.2, 'Jupiter': 69911.0, 'Saturn': 58
232.2}
```

Here we see that the type of **solar_planets** is called 'dict', which means it's a data structure of dictionary.

Recall: lists using “[ ]”, tuples using “( )”, sets using “{ }” as well

Syntax: Dict_Name{key1:value1, key2:value2, key3:value3, …}

The nice thing about a dictionary is that you can access the values using the keys, just like looking up a word in a dictionary. Recall the way to access an element in a list using indices: e.g., `mylist[2]`, we also use “[ ]” to access values in a Python dictionary:

```
print(solar_planets['Venus'])
```

```
6052.1
```

But it’s not exactly your home dictionary, you can change the values for a specific key

```
solar_planets['Earth'] = 6380.0 # change the Earth radii to 6380.0
```

After you’ve done that, the dictionary is changed:

```
print(solar_planets)
```

```
{'Mercury': 2440.0, 'Venus': 6052.1, 'Earth': 6380.0, 'Mars': 3389.2, 'Jupiter': 69911.0, 'Saturn': 58232.2}
```

Adding key-value pairs to your dictionary

```
solar_planets['Uranus'] = 25362.0  # add Uranus to the dict
solar_planets['Neptune'] = 24622.3 # add Neptune to the dict
print(solar_planets)               # print out the new dict
```

```
{'Mercury': 2440.0, 'Venus': 6052.1, 'Earth': 6380.0, 'Mars': 3389.2, 'Jupiter': 69911.0, 'Sa
turn': 58232.2, 'Uranus': 25362.0, 'Neptune': 24622.3}
```

Remove key-value pairs to your dictionary

```
del solar_planets['Earth'] # remove the key 'Earth' from the dictionary - the corresponding value is also removed
print(solar_planets)       # print out the new dict
```

```
{'Mercury': 2440.0, 'Venus': 6052.1, 'Mars': 3389.2, 'Jupiter': 69911.0, 'Saturn': 58232.2, 'Uranus': 25362.0, 'Neptu
ne': 24622.3}
```

A couple of useful methods for the 'dict' object

- keys(): return the keys in a dictionary
- values(): return all the values in a dictionary

```
In [7]: solar_planets.keys()
```

```
Out[7]: dict_keys(['Mercury', 'Venus', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'])
```

```
In [22]: solar_planets.values()
```

```
Out[22]: dict_values([2440.0, 6052.1, 3389.2, 69911.0, 58232.2, 25362.0, 24622.3])
```

Recall: use the list( ) function to generate a list from the keys or values of a dictionary

```
In [8]: planets = list(solar_planets.keys())
        print(planets)
```

```
['Mercury', 'Venus', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

Basic Python Code Structures

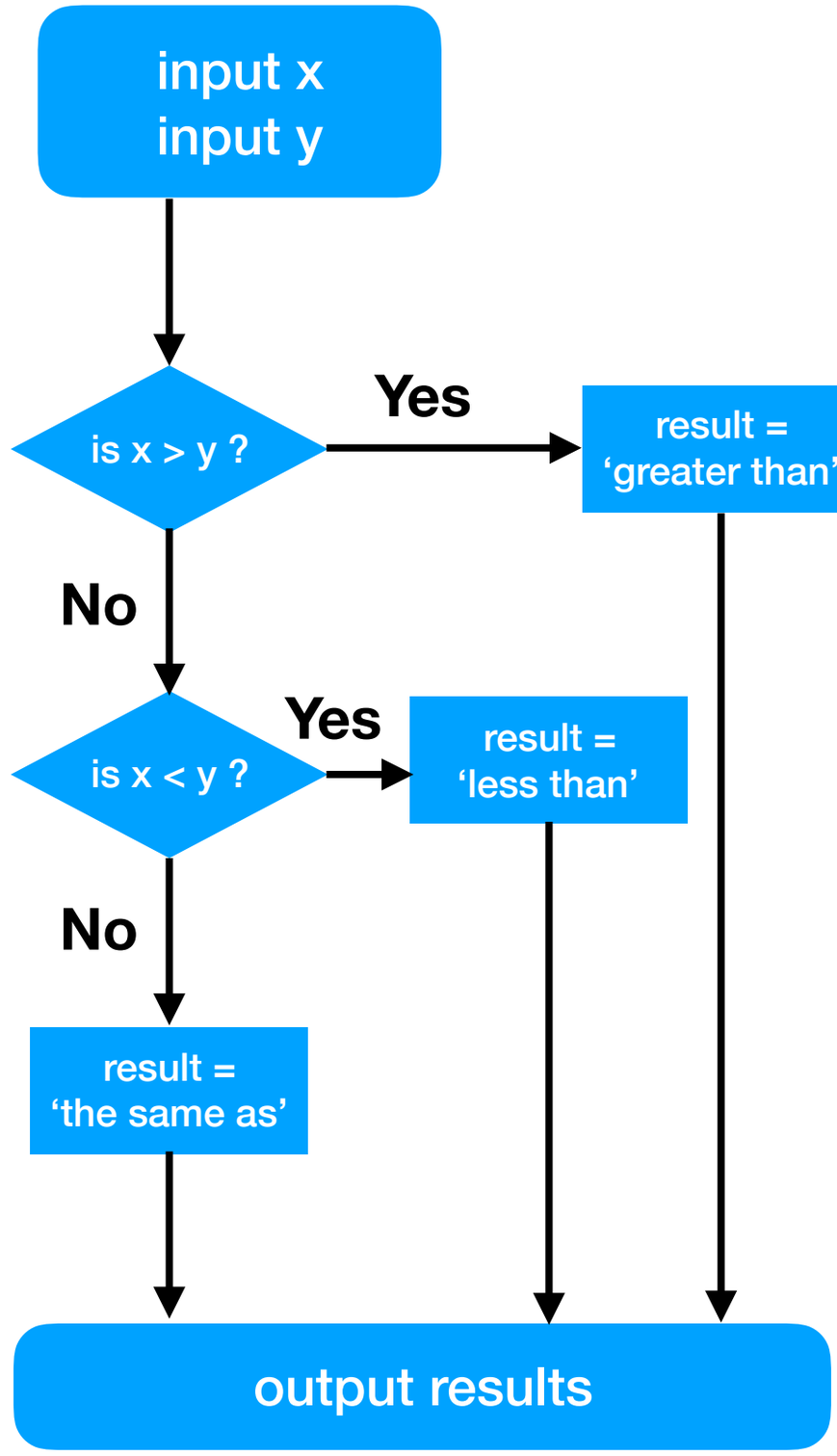
```
# Here's a simple Python code

# Code black 1: asking for inputs
x = input('Input number x: ')
y = input('Input number y: ')

# Code black 2: decide which is larger
if (x > y):
    result = 'greater than'
elif (x < y):
    result = 'less than'
else:
    result = 'the same as'

# Code block 3: print out the results
print('Results: x is',result,'y')
```

```
Input number x: 2
Input number y: 1
Results: x is greater than y
```

Code block 1

Code block 2

Code block 3



Basic Python Code Structures: Indentations

- Python uses **indentation** to define the *code blocks* and this also makes the code readable.
- In a Python code, each block starts with a condition statement (if this is True) terminated with a '**:**'
- A typical Python program looks like this:

```
program statement
Function Definition 1
Function Definition 2
…
Code block 1 condition statement:
    block 1 statement 1
    block 1 statement 2
    block 1 statement 3
    ……
    Code block 2 condition statement:
        block 2 statement 1
        block 2 statement 2
        Code block 3 condition statement:
            block 3 statement 1
            block 3 statement 2
            Code block 4 condition statement: block 4 statement
        block 2 statement 3
        block 2 statement 4
        ……
    block 1 statement 4
    block 1 statement 5
(end of program)
```

A couple of rules:

- Use only spaces to indent your code. You could use tabs as well, just make sure use only ONE type of spaces in one code
- Indentation: typically **four** spaces
- A statement can be continued on to the next line with the backslash sign "\" and the indentation of that code block
- If a code block has only one statement, it may be placed in the same line as the colon (see the Code block 4 on left)
- The reserved word (command) "break" breaks you out of the code block. Use with caution!
- The reserved word (command) "pass" can be used to stand in for a code block by doing nothing

Basic Python Code Structures: Condition Statements

Condition statements are used in your programs to control the flow of the execution. They usually include a program control statement (if, for, while) together with relational and/or logical operations.

Relational operators

Here are frequently used relational operators in Python:

- "==" means "equals", for example: a == b (does variable a equal to variable b)?
- "!=" means "does not equal", for example: a != b (does variable a not equal to variable b)?
- "<" means "less than", for example, 1 < 2 gives **True**, while 3 < 2 gives **False**
- "<=" means "less than or equal to", for example, 1 <= 2 gives **True**, while 2 <= 2 also gives **True**
- ">" means "greater than", for example, 1 > 2 gives **False**, while 3 > 2 gives **True**
- ">=" means "Greater than or equal to", for example, 1 >= 2 gives **False**, while 2 >= 2 gives **True**

Logical operators

Here are frequently used logical operators in Python:

- "**and**" means “both are true", for example: A **and** B (if both A and B are true, the result is **True**)
- "**or**" means “either is true", for example: A **or** B (if either A or B is true, the result is **True**)
- "**not**" means “take the opposite", for example, **not** A gives **True**, while A is **False**

and	TRUE	FALSE
TRUE	T	F
FALSE	F	F

or	TRUE	FALSE
TRUE	T	T
FALSE	T	F

not	TRUE	FALSE
TRUE	F	T

Python Operator Precedence (selected)

high

low

Priority	Operator	Description
1	`()`	Parentheses (grouping)
2	`**`	Exponentiation
3	`*, /, %`	Multiplication, division, remainder
4	`+, -`	Addition, subtraction
5	`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
6	`not x`	Boolean NOT
7	`and`	Boolean AND
8	`or`	Boolean OR

For example

x = 7 + 2 * 3

x = (7 + 2) * 3

x = 7 + 2 % 3 ** 2

x = 7 + (2 * 3) ** 2

x = 7 / 3 + (2 * 3) ** 2 and y > 1

Better be more specific:

x = (7 / 3 + (2 * 3) ** 2) and (y > 1)

It's makes your programs more readable

Python Operator Precedence (full)

Priority	Operator	Description
1	`()`	Parentheses (grouping)
2	*f*`(args...)`	Function call
3	*x*`[index:index]`	Slicing
4	*x*`[index]`	Subscription
5	*x.attribute*	Attribute reference
6	`**`	Exponentiation
7	`~`*x*	Bitwise not
8	`+`*x*`, -`*x*	Positive, negative
9	`*, /, %`	Multiplication, division, remainder
10	`+, -`	Addition, subtraction
11	`<<, >>`	Bitwise shifts
12	`&`	Bitwise AND
13	`^`	Bitwise XOR
14	`|`	Bitwise OR
15	`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
16	`not` *x*	Boolean NOT
17	`and`	Boolean AND
18	`or`	Boolean OR
19	`lambda`	Lambda expression

Basic Python Code Structures: Condition Statements

Here's a simple example of condition statements

In [*]:
```
# This is a good code judging your favorite food

food = input("What's your favorite food? ")

if food == 'garlic':
   print('I think',food,'is super yucky')
else:
   print('I think',food,'is super yummy')
```

Condition statement

```
What's your favorite food?
```

Here's are a couple more example of condition statements

In [25]:
```
# here are some examples for conditions statements (aka. logic operations)
print('good'=='bad')
print('good'!='bad')
print(3.14 > 20)
print(4.5 <= 22)
print( ('Earth'=='Mars')and(6380.0>=3350.0) )
print( ('Earth'=='Mars')or(6380.0>=3350.0) )
```

```
False
True
False
True
False
True
```

Question: What is the answer if you do: 'Earth' > 'Mars'? Why is that? Can you figure out the reason by yourself?

Python Program Control: the “if” statements

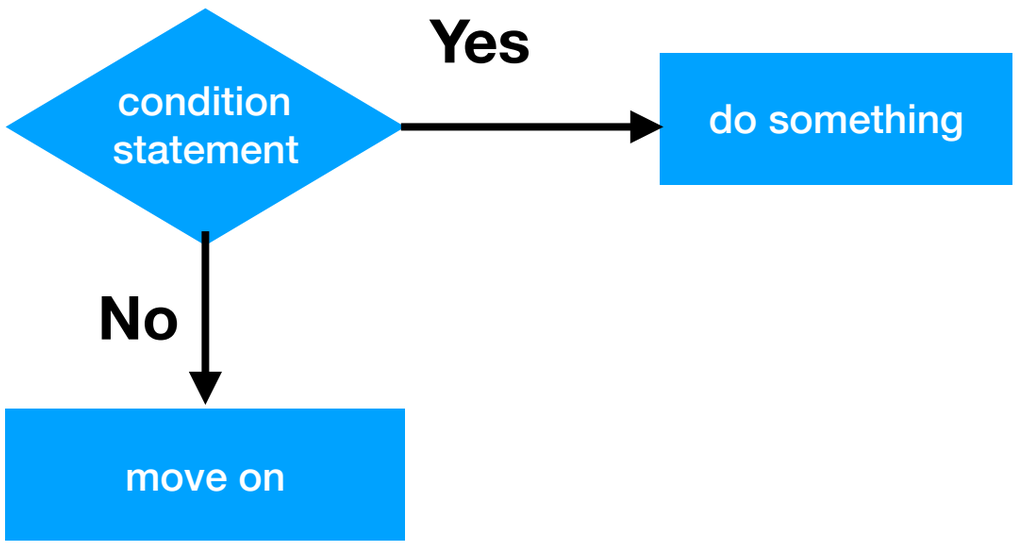
Syntax of an “if”-statement

```
if condition statement:
    do something
```

Indentation: 4 spaces

How the “if”-statement works:

- First evaluate the “`condition statement`”
- When the “`condition statement`” is **True**, execute the indented code “do something”
- When the “`condition statement`” is **False**, skip the indented code and don’t “do something”



Python Program Control: the “if-else” statements

Syntax of an “if-else”-statement

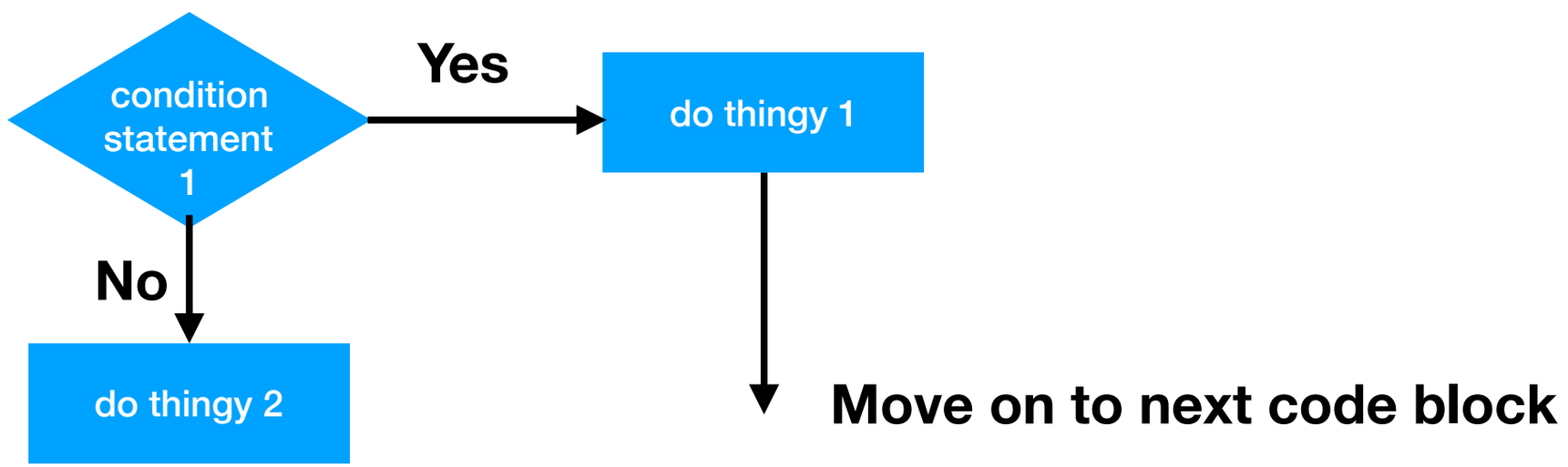
```
if condition statement:
    do thingy 1
else:
    do thingy 2
```

Indentation: 4 spaces

Note:

How the “if-else”-statement works:

- First evaluate the “`condition statement`”
- When the “`condition statement`” is **True**, execute the indented code “do thingy 1”
- When the “`condition statement`” is **False**, execute the indented code “do thingy 2”

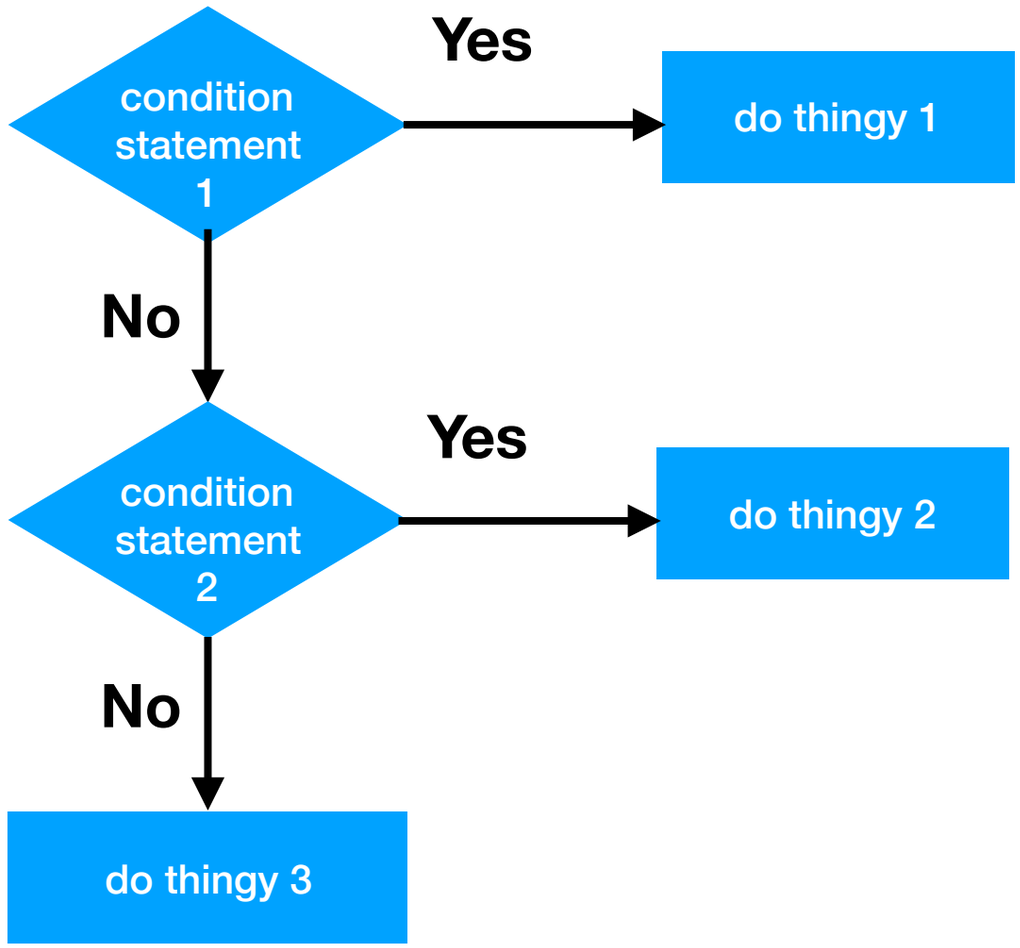


Python Program Control: the “if-elif-else” statements

Syntax of an “if-else if-else”-statement

```
if condition statement 1:
    do thingy 1
elif condition statement 2:
    do thingy 2
else:
    do thingy 3
```

Indentation: 4 spaces



How the “if-elif-else”-statement works:

- First evaluate the “`condition statement 1`”
- When the “`condition statement 1`” is **True**, execute “do thingy 1” , them move on
- When the “`condition statement 1`” is **False**, evaluate the “`condition statement 2`”, if it’s **True**, execute “do thingy 2”; if it’s **False**, execute “do thingy 3” and then move on

Let's see a couple of examples for the "if" statements:

- **if** and **if-else** statements

In [26]:
```
# a simple example of if-loop

N = 3 # define an integer variable N with a value of 3
if (N < 0): # is N less than zero?
    print("N is negative!")
# if N is not less than zero, the print() part inside the "if" code block is not executed
# so there's no output from this statement. If you change N to be -3, you should get the output saying
# "N is negative!":

N = -3 # define an integer variable N with a value of -3
if (N < 0): # is N less than zero?
    print("N is negative!")
else:
    print('N is positive!')
```

```
N is negative!
```

- **if-elif-else** statement

In [32]:
```
latitude  = 22.4 # define a variable 'latitude'

if (latitude < 24):
    print("Tropical region")
elif (latitude > 24 and latitude < 66):
    print("Temperate region")
else:
    print("Polar region")
```

```
Tropical region
```

- **if-elif-else** statement and the **pass** statement

In [33]:
```
mylist=['jane','josh','sid','geoff'] # define a list

if 'susie' in mylist:  # if the string "susie" is in the list, then
    pass # don't do anything

if 'susie' not in mylist:
    print ('call susie and apologize!')
    mylist.append('susie')
elif 'george' in mylist: # if first statement is false, try this one
    print ('susie and george both in list')
else: # if both statements are false, do this:
    print ("susie in list but george isn't")

print(mylist)
```

```
call susie and apologize!
['jane', 'josh', 'sid', 'geoff', 'susie']
```

"membership" operator

Membership operators are operators used to validate the membership of a value. It test for membership in a sequence, such as strings, lists, or tuples.

- **if-elif-elif-elif-else** statement

In [31]:
```
exam = 82.0 # my final exam score

if exam >= 90.0:   # if scored greater than or equal to 90, I get 'A'
    Grade = 'A'
elif exam >= 80.0: # if scored greater than or equal to 80, I get 'B'
    Grade = 'B'
elif exam >= 70.0: # if scored greater than or equal to 70, I get 'C'
    Grade = 'C'
elif exam >= 60.0: # if scored greater than or equal to 60, I get 'D'
    Grade = 'D'
else:              # Otherwise, I get 'Fail'
    Grade = 'Fail'

print('Your final grade is', Grade)
```

```
Your final grade is B
```

Python Program Control: the “while” loops

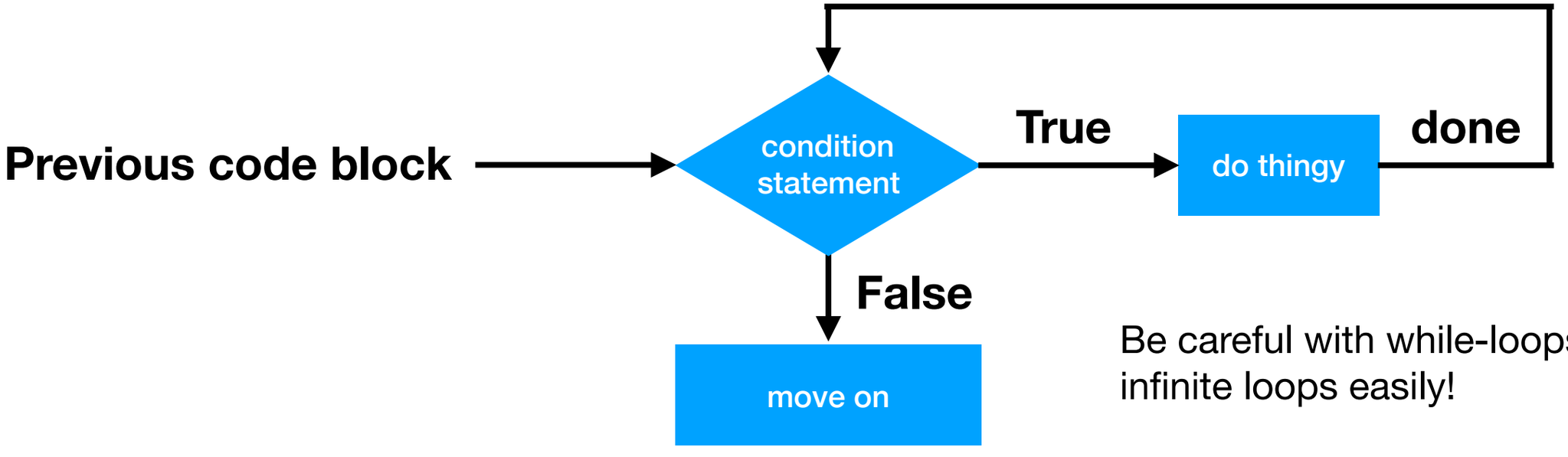
Syntax of a “while”-loop

```
while condition statement:
    do thingy here
```

Indentation: 4 spaces

How the “while”-loop works:

- **Step1:** First evaluate the “`condition statement`”
- Step 2: When the “`condition statement`” is **True**, execute the indented code “do thingy here”
- Step 3: When finishes “do thingy here”, go back to the while statement and test “`condition statement`” **again**, if it is still True, go back to Step 2; otherwise, move on to the next code block (leave the while-loop)



Be careful with while-loops, you can get infinite loops easily!

Let's see an example of a typical "while" loop:

In [34]:
```
mylist=['jane','josh','sid','geoff'] # define a list

while 'susie' not in mylist: # check if 'susie' is NOT in the list
    print('call susie and apologize!') # if not: first print
    mylist.append('susie')             #         then append 'susie' to mylist

print(mylist)
```

```
call susie and apologize!
['jane', 'josh', 'sid', 'geoff', 'susie']
```

This while-loop only gets executed once

In [35]:
```
N = 100 # define an integer variable N to be 100
threshold = 90 # define some threshold to be 90 (also integer)

# while the condition in the () is true, the program will continue
while (N > threshold):  # is N greater then threshold?  if so, execute the indented code block
    print (N)  # what it says
    N = N-1    # decriment N by one, some people uses N-=1, which is the same thing

print ('and the final value is: ',N) # now we're done we'll see what is left of N.
```

```
100
99
98
97
96
95
94
93
92
91
and the final value is:  90
```

This while-loop only gets executed multiple times

Question: Why the final value is 90 instead of 91?

Python Program Control: the “For” loops

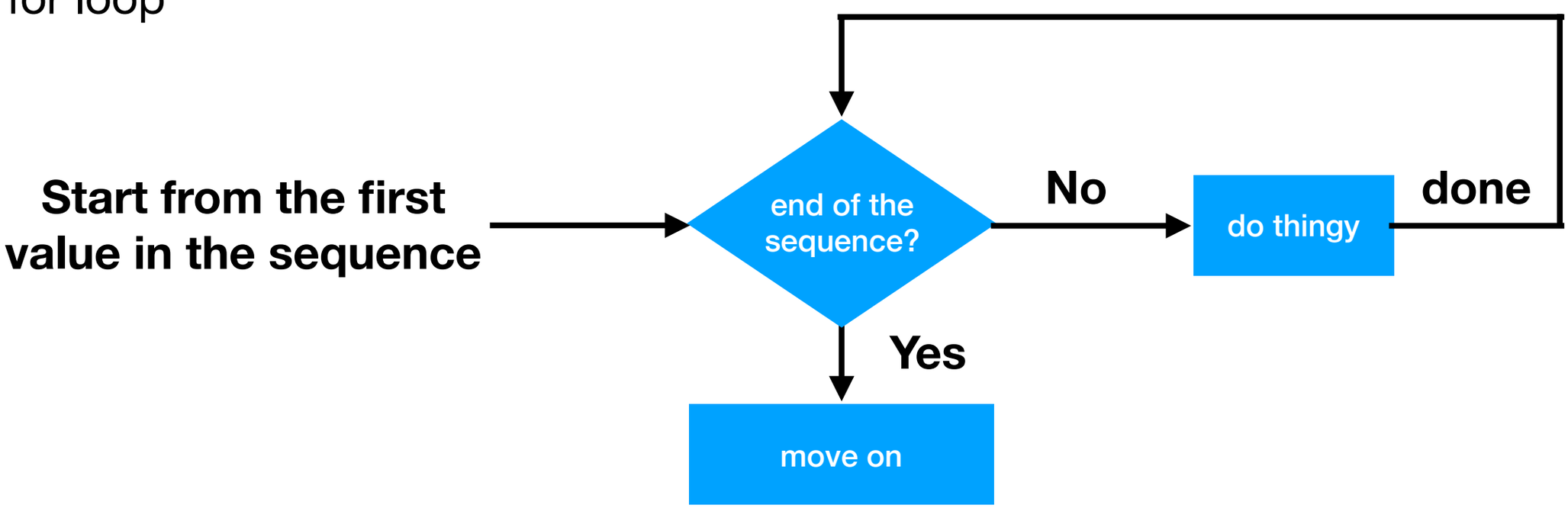
Syntax of a “for”-loop

```
for a variable in a sequence:
    do thingy here
```

Indentation: 4 spaces

How the “for”-loop works:

- **Step1:** assign the first value in the sequence to the looping variable
- Step 2: execute the indented code “do thingy”
- Step 3: assign the next value in the sequence to the looping variable, and go back to Step 2
- Step 4: when all the values in a list are stepped through, move on and leave the for loop



Let's see a couple of examples for typical "for" loops:

e.g., "for"-loops usually work well with the `range()` function we learned before

In [36]:
```
# make a list with the range() function
numbers = range(10) #creates a list of numbers 0 to 10 stepping by 1

# step through the list assigning each element to the variable, n
for n in numbers: # n is assigned to each element in turn
    print (n, n*n, n**n ) # or we could have written print n**2
```

```
0 0 1
1 1 1
2 4 4
3 9 27
4 16 256
5 25 3125
6 36 46656
7 49 823543
8 64 16777216
9 81 387420489
```

`range(10)` here means `range(0,10,1)`

"for"-loops also work well lists, tuples, sets and dictionaries, this is a very nice feature of Python

In [37]:
```
mylist=['jane','josh','sid','geoff'] # define a list

for name in mylist: # create a variable called 'name', and assign each element in 'mylist' to name
    print(name)     # print out the variable 'name'
```

```
jane
josh
sid
geoff
```

`for name in mylist:` go through all the elements in a list

"name" is a temporary variable

Here's an example of a for-loop stepping through the keys of a dictionary:

In [38]:
```
# recall the dictionary 'solar_planets' we used in the beginning of the lecture, re-define it here
solar_planets = {'Mercury':2440.0, 'Venus':6052.1, 'Earth':6371.5, 'Mars':3389.2, 'Jupiter':69911.0, 'Saturn':58232.2}

# print out the list generated
print ('List of keys in solar_planets: ',list( solar_planets.keys() ) ) # here you need the list() function
# notice the slightly fancier form of the print statement above.

# now step through that list,
   # assigning each element to the variable key
for key in solar_planets.keys():  # just use the list generator (we don't need the list() function)
    print ('The radius of planet', key, 'is', solar_planets[key], 'km') # print out the variable, key
```

```
List of keys in solar_planets:  ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn']
The radius of planet Mercury is 2440.0 km
The radius of planet Venus is 6052.1 km
The radius of planet Earth is 6371.5 km
The radius of planet Mars is 3389.2 km
The radius of planet Jupiter is 69911.0 km
The radius of planet Saturn is 58232.2 km
```

- **"solar_planets.keys( )" generates a sequence contains all the keys in the dictionary named "solar_planets"**
- **Variable "key" here is a temporary variable, you can use other names as well**
- **The variable "key" get assigned as one key of dictionary each time**
- **After the for loop is done, the variable "key" is destroyed by Python and no longer has any value**

Python Program Control: “Nested” loops

In Python you can combine the for-loops, if-statements and/or while-loops together to develop complicated codes for executing advance algorithms. Let’s take a look at an example here

In [39]:
```
solar_planets = {'Mercury':440.0, 'Venus':737.1, 'Earth':288.5, 'Mars':210.2, 'Jupiter':110.0, 'Saturn':81.2}

habitable = False

for key in solar_planets.keys():
    temperature = solar_planets[key]
    temperature = temperature - 273.1
    solar_planets[key] = temperature
    if (temperature <= 100) and (temperature >= -100):
        habitable = True
        print('Planet',key,'is habitable with a Surface Temperature of ',temperature)

print(solar_planets)
```

loop