SEE1002
Introduction to Computing for Energy and Environment

Part 2: Elements of Python programming

Sec. 4: Repeating tasks 1: Loops

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Data and variables

Section 2: Elementary data structures

Section 3: Branching

Section 4: Loops

Section 5: Functions

Part 3: Basic Python programming

Section 1: Structure of a Python program

Section 2: Input and output

Section 3: Modules

Section 4: Good programming practices

Part 4: Python for science and engineering

Section 1: Vectors, matrices and arrays

Section 2: NumPy and SciPy

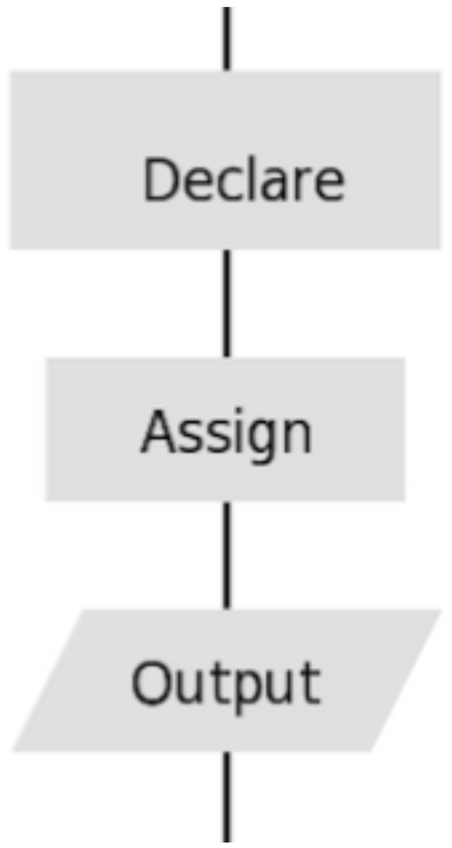
Outline

1. Motivation
2. `while` loop
3. `for` loop

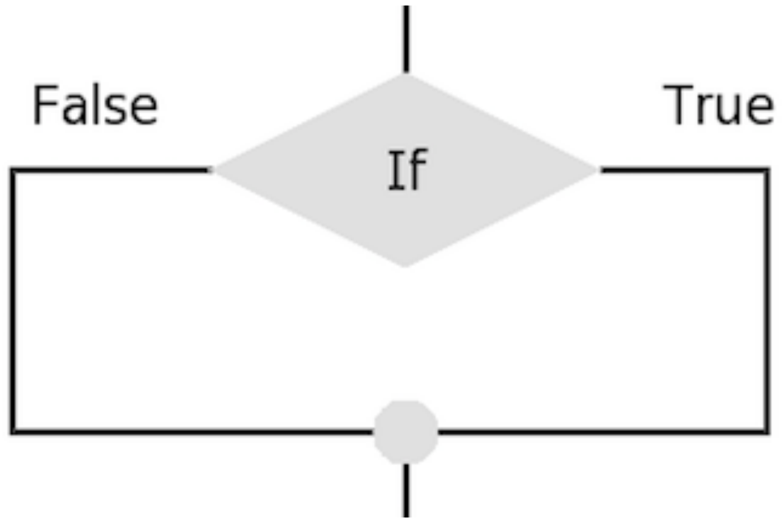
1. Motivation

Recap

We have been introduced to two basic types of program flow:



Linear flow



Branching

Limitations

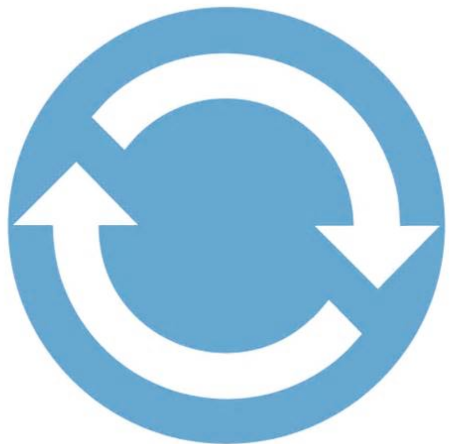
- We can get pretty far with what we've learned so far.
- But it will take us a lot of work to solve all but the simplest problems!
- This is because we yet to introduce a method for repeating tasks.

Example 1: manual counting

How would you write a program to print integers from 1 to 3? How would your approach change if you had to consider 1 to 20? 1 to 1000000?

Repeating tasks efficiently

- Most problems solved on a computer differ a lot from the sort of problems we've examined so far.
- Generally the computer spends most of its time carrying out the same set of tasks. Remember that computers are good at doing things quickly and accurately!
- By contrast humans are error prone. So we want to write simple programs that allow computers to do what they're good at.
- We need an efficient way of repeating tasks. The easiest way is with a loop.



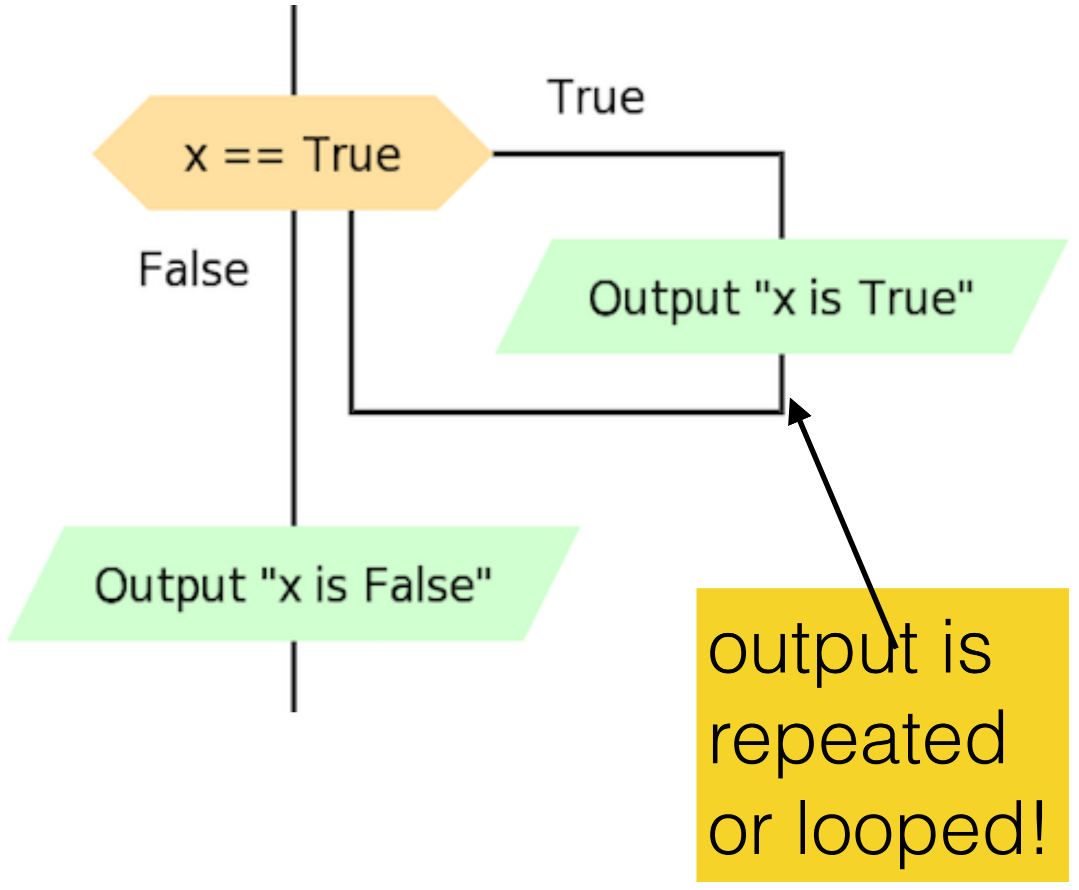
2. `while` loop

Motivation

- The simplest way of repeating a task is to **keep doing the same thing until the situation changes (i.e. while a certain condition holds)**
- *Examples*
  - Keep going straight until you reach the corner.
  - Boil the pasta until it becomes tender
  - Apply for a job until you're hired

Overview

The `while` loop executes a series of statements while a condition is true.



```
x=True
while x==True:
    print('x is True')
else:
    print('x is False')
```

Flowgorithm

Python

Syntax of the `while` statement

More precisely, the `while` statement has the following syntax:

```
while <expression>:
    statement 1.1
    […]
else:
    statement 2.1
    […]
```

executed if *`expression`*`==True`

executed if *`expression==False`*

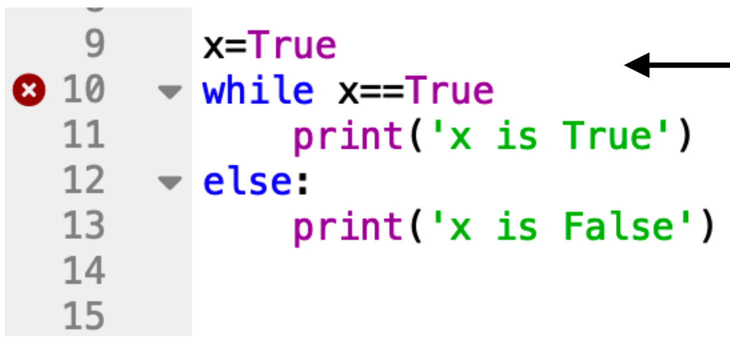
Comments on the syntax

- The syntax is essentially identical to that of `if-else`. So the same rules need to be observed:

  1. Colon follows `while` and `else`

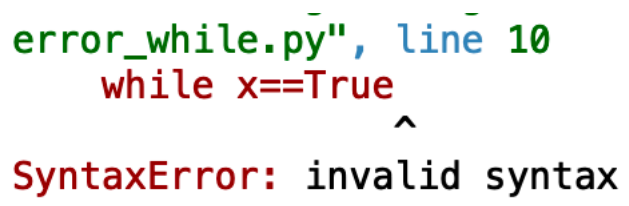
  2. The statements to be executed following `while` and `else` must be indented by the same amount (typically 4 spaces).

- If these rules are not followed, the program will not behave as expected:



```
 9   x=True
10   while x==True
11       print('x is True')
12   else:
13       print('x is False')
14
15
```

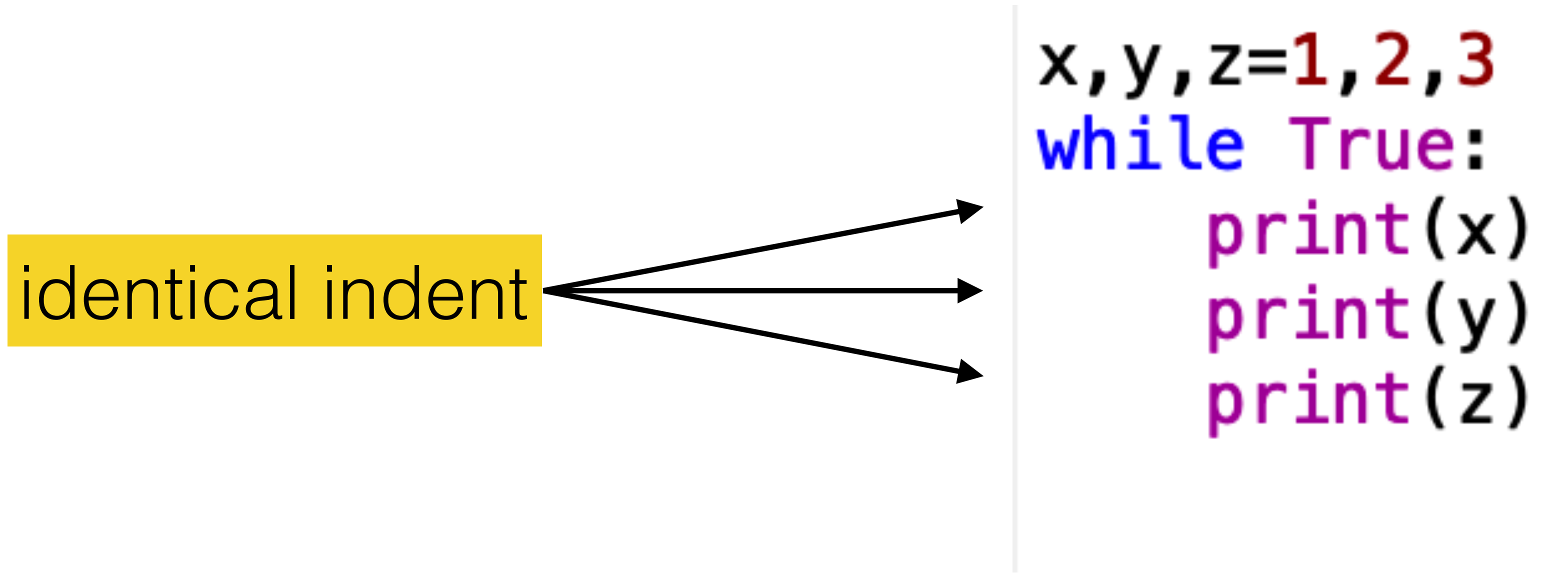
← missing colon



```
error_while.py", line 10
    while x==True
                ^
SyntaxError: invalid syntax
```

Multiple statements

- As with `if-else`, multiple statements can be executed for the true and false conditions.
- Statements that are executed sequentially must be indented by the same amount.



```
x,y,z=1,2,3
while True:
    print(x)
    print(y)
    print(z)
```

Types of `while` loops

There are essentially 3 different types of `while` loops:

i. infinite loop

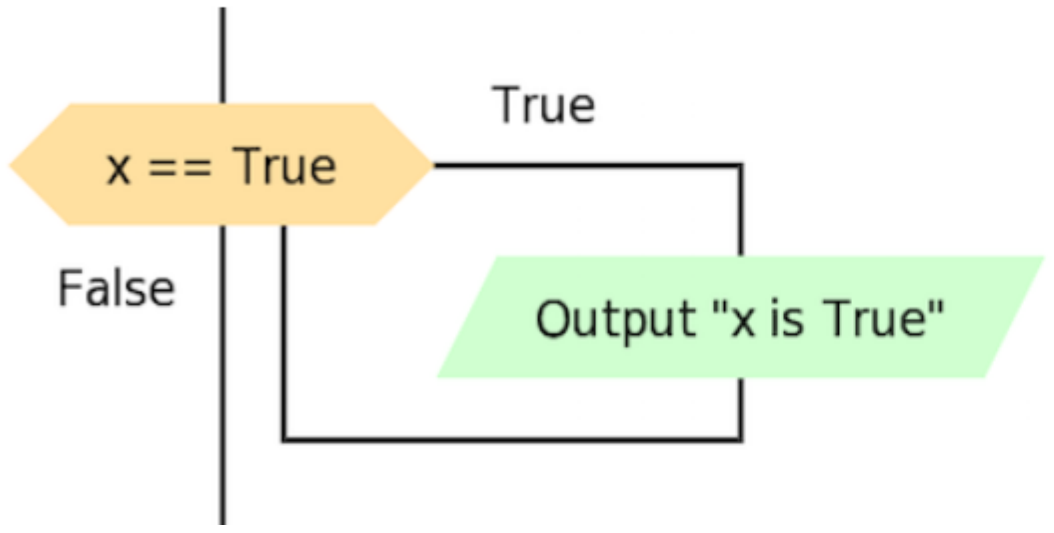
II. finite loop

III. loop over index

i. Infinite loop

- An infinite loop is constructed using a condition that’s always true.
- The only way to get out of a infinite loop is to hit `control-c` or the stop button.
- An infinite loop is one of the first things learned by any programming student. But its practical applications are fairly limited.

Implementation

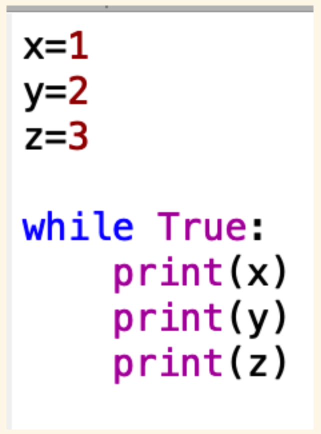


```
x=True
while x==True:
    print( 'x is True' )
```

```
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
x is True
```

Example 2: infinite loop with multiple statements

Consider this Python program. What do you think it does?



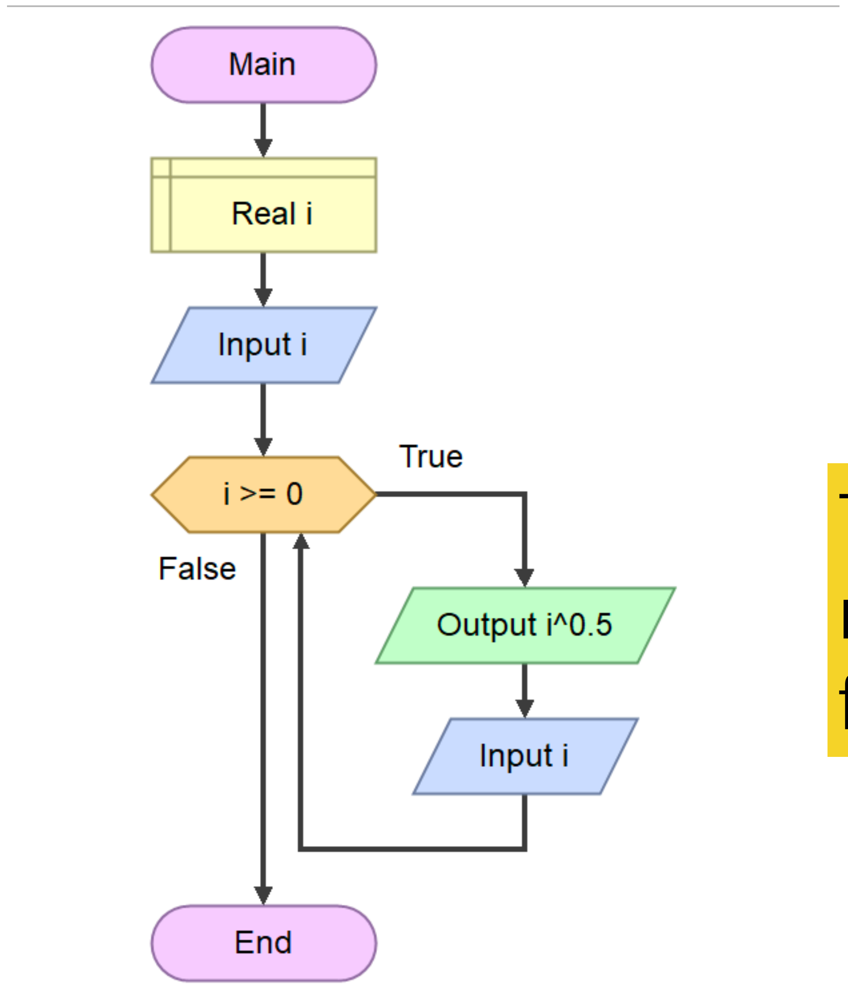
```
x=1
y=2
z=3

while True:
    print(x)
    print(y)
    print(z)
```

ii. Finite loop

- A much more useful `while` loop is one in which the test condition isn't always true.
- Hence the loop only runs for a finite number of times.
- This is the most common use of the `while` loop. One often wants to continue testing a condition until it has changed.

Implementation

Typically we stop looping once a condition is met.

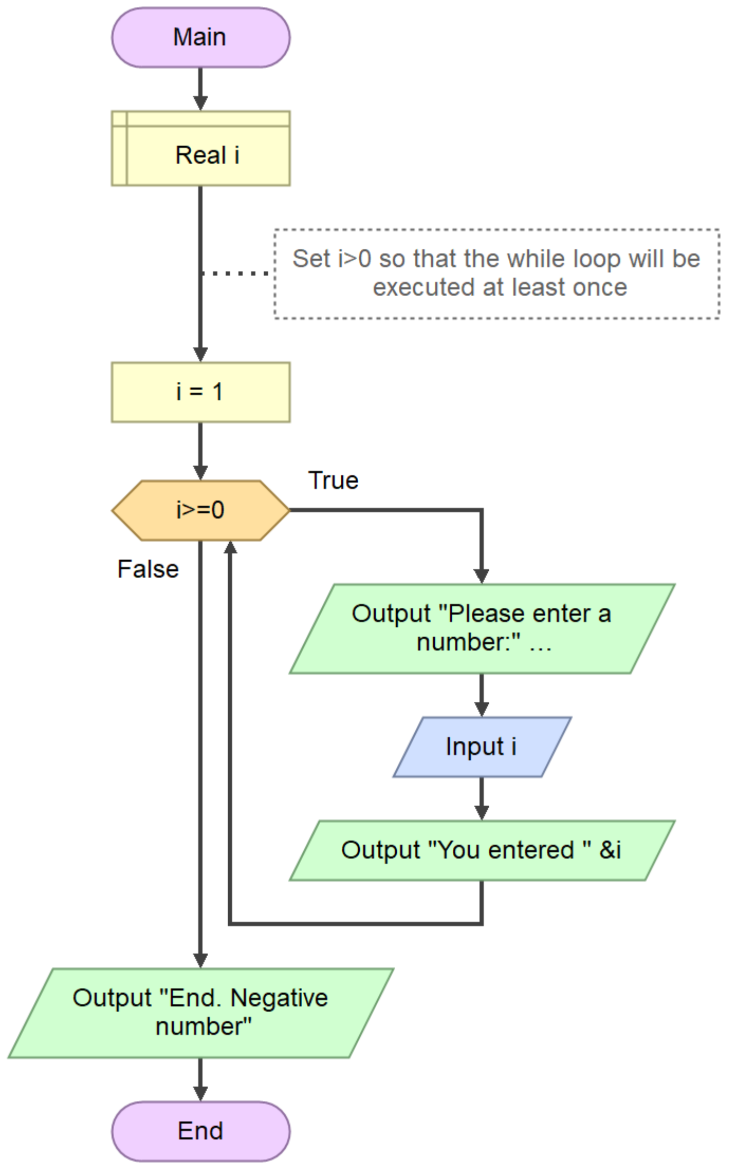


```
i=float(input('please enter a number: '))
while i >=0:
    print(i**0.5)
    i=float(input('please enter a number: '))
```

The program exits the loop once a negative number is input, i.e., the loop is finite

Repeated testing

The most common finite loop is the one we just saw, in which we repeatedly test for a condition. To save typing, we can use an arbitrary value for the first test.

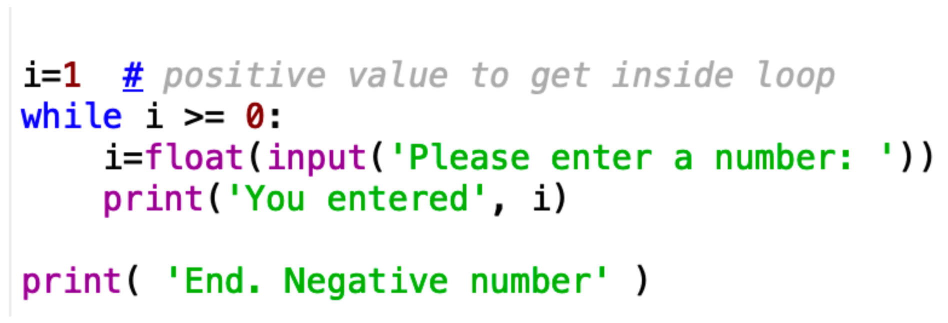

Main
Real i
Set i>0 so that the while loop will be executed at least once
i = 1
i>=0
True
False
Output "Please enter a number:" …
Input i
Output "You entered " &i
Output "End. Negative number"
End

```
i=1  # positive value to get inside loop
while i >= 0:
    i=float(input('Please enter a number: '))
    print('You entered', i)

print( 'End. Negative number' )
```

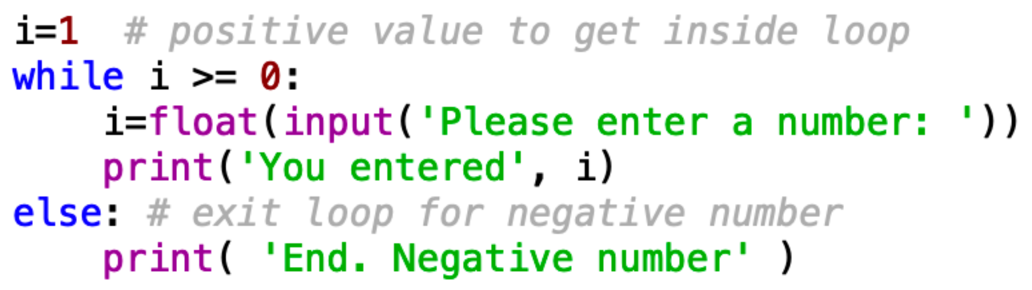
```
Please enter a number: 4.3
You entered 4.3

Please enter a number: -1.2
You entered -1.2
End. Negative number
```

Alternative version

We can rewrite our program using an else block to highlight what happens when a negative number is input. Both versions are equivalent.



```python
i=1  # positive value to get inside loop
while i >= 0:
    i=float(input('Please enter a number: '))
    print('You entered', i)

print( 'End. Negative number' )
```

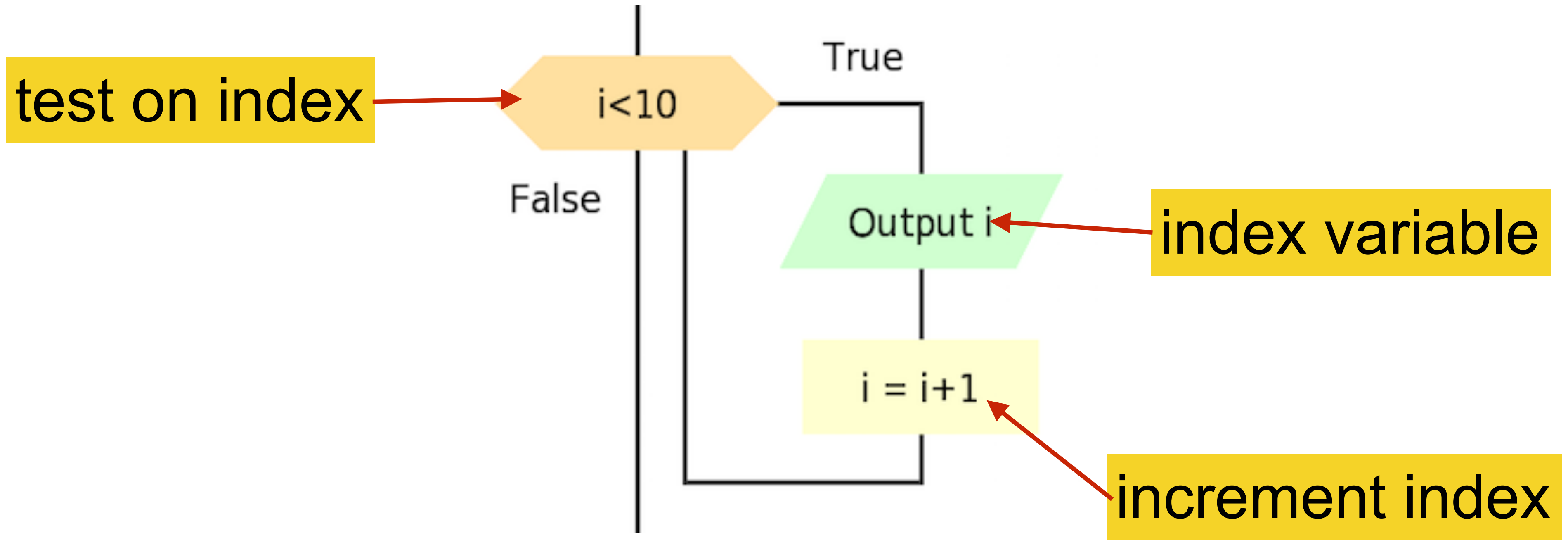


```python
i=1  # positive value to get inside loop
while i >= 0:
    i=float(input('Please enter a number: '))
    print('You entered', i)
else: # exit loop for negative number
    print( 'End. Negative number' )
```

iii. Loop over index

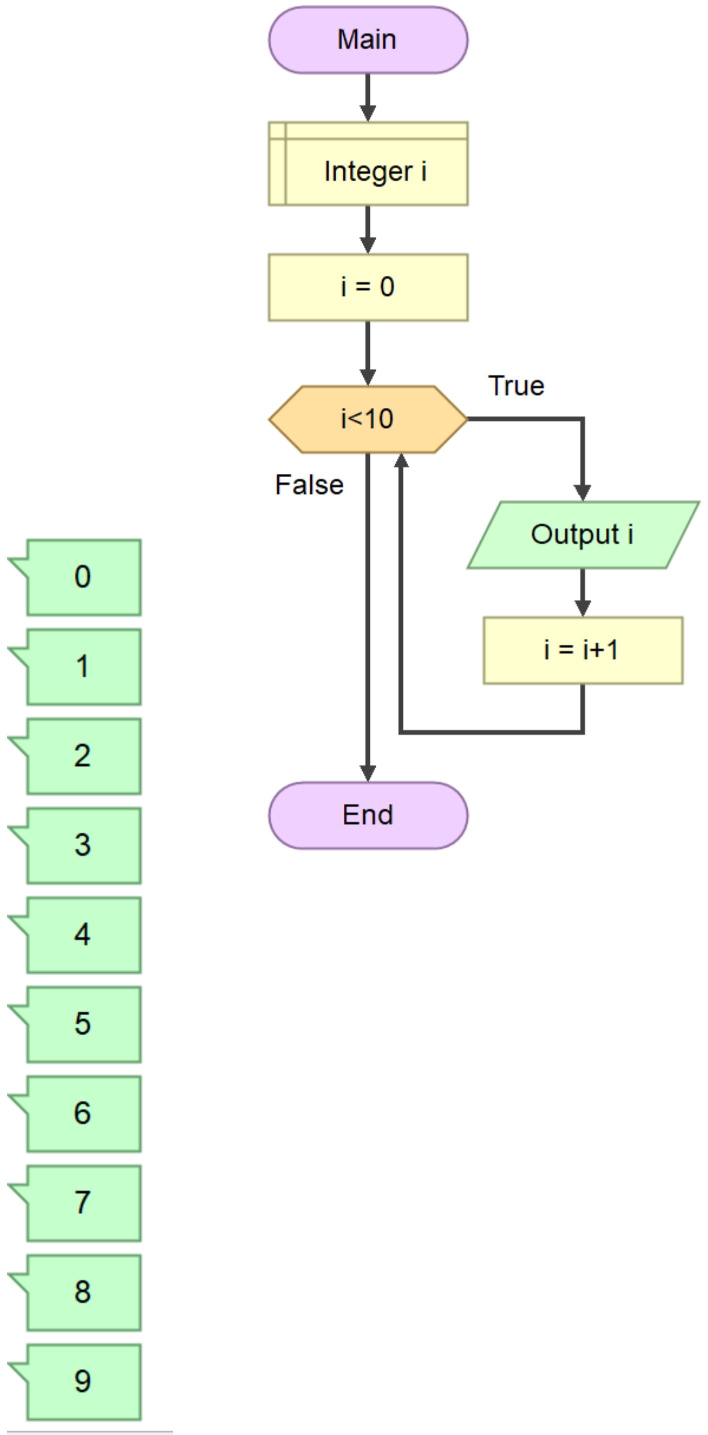
Instead of repeated testing, we can perform a specified number of loops. To this we need to count the number of loops using an index or counter variable.

- The index variable defines the condition under which the `while` loop is executed.
- The index variable must be increased or incremented inside the loop



Python implementation

The Python implementation exactly follows the Flowgorithm one almost identically.



```python
i=0
while i<10:
    i = i + 1
    print( i )
```

```
1
2
3
4
5
6
7
8
9
10
```

Example 3: return on an investment

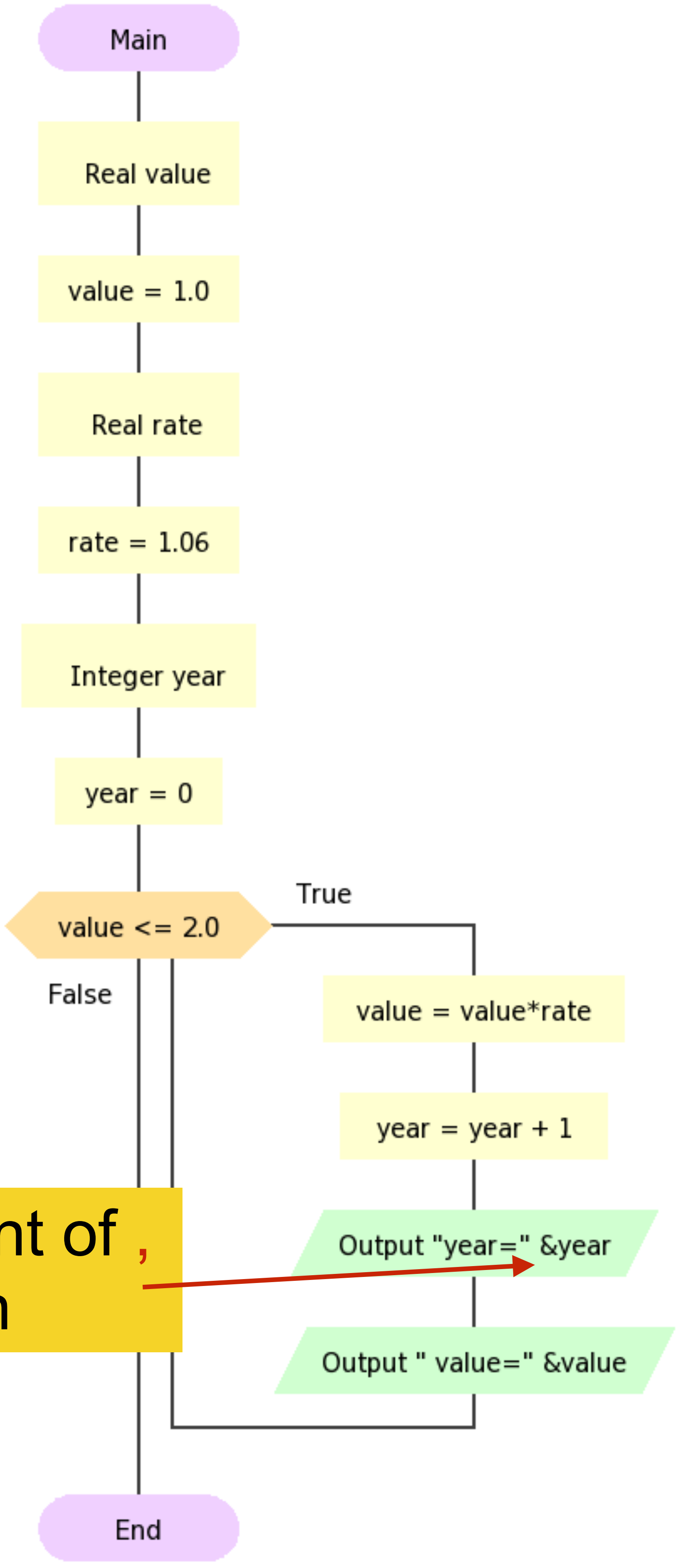
- There are simple formulas for calculating compound interest and the return on an investment, but let's pretend that we don't know them.
- Use a `while` loop to calculate the number of years required for an investment to more than double in value given an interest rate of 6%. Output the value of the investment after each year.
- *We will use this example to illustrate various features of Python!*

Example 3: approach

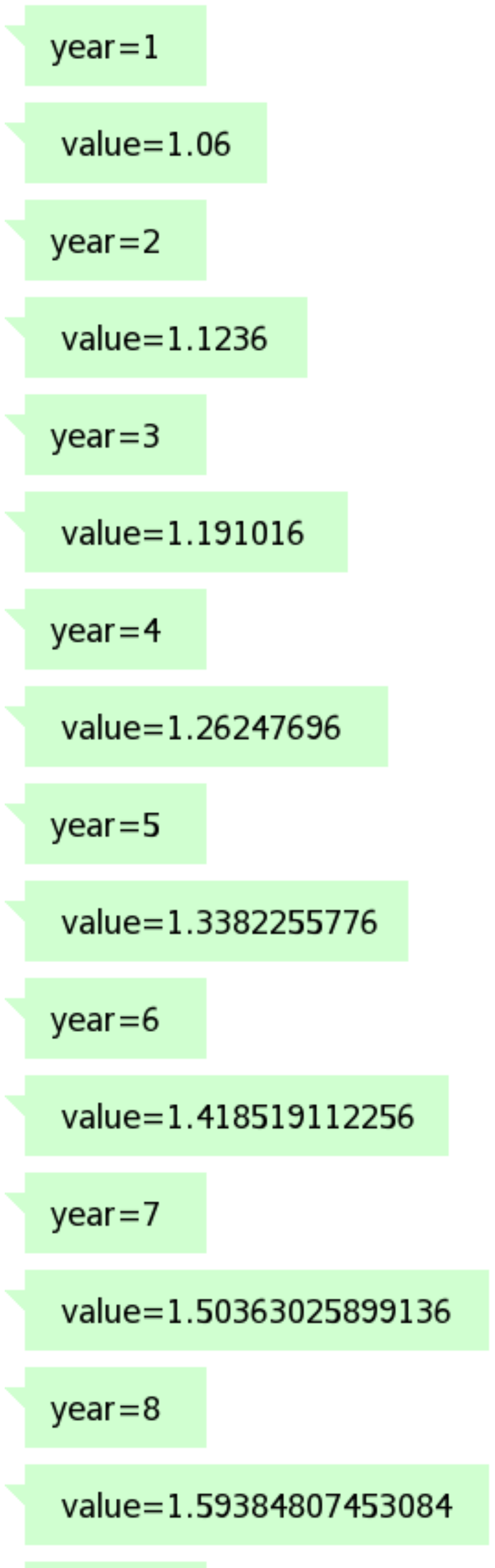
How do we use a `while` loop? Continue calculating the value of the investment so long as the value hasn't increased by more than a factor of 2. *This is a finite loop in which we test on a condition.*

- Needed variables:
  - `value` (current value of investment)
  - `years` (number of years elapsed)
- Needed constant
  - `rate` (interest rate +1)

Implementation in Flowgorithm



Equivalent of , in Python



```
year=1
value=1.06
year=2
value=1.1236
year=3
value=1.191016
year=4
value=1.26247696
year=5
value=1.3382255776
year=6
value=1.418519112256
year=7
value=1.50363025899136
year=8
value=1.59384807453084
```

Basic Python solution

```
value = 1.0
rate  = 1.06
year  = 0

while value <= 2.0:
    value = value * rate
    year = year + 1
    print( 'year: ',year,'value:',value )
else:
    print( 'It takes',year,'years for the investment to double in value' )
```

Final output message

```
year:  1 value: 1.06
year:  2 value: 1.1236
year:  3 value: 1.191016
year:  4 value: 1.26247696
year:  5 value: 1.3382255776
year:  6 value: 1.41851911226
year:  7 value: 1.50363025899
year:  8 value: 1.59384807453
year:  9 value: 1.689478959
year:  10 value: 1.79084769654
year:  11 value: 1.89829855834
year:  12 value: 2.01219647184
It takes 12 years for the investment to double in value
```

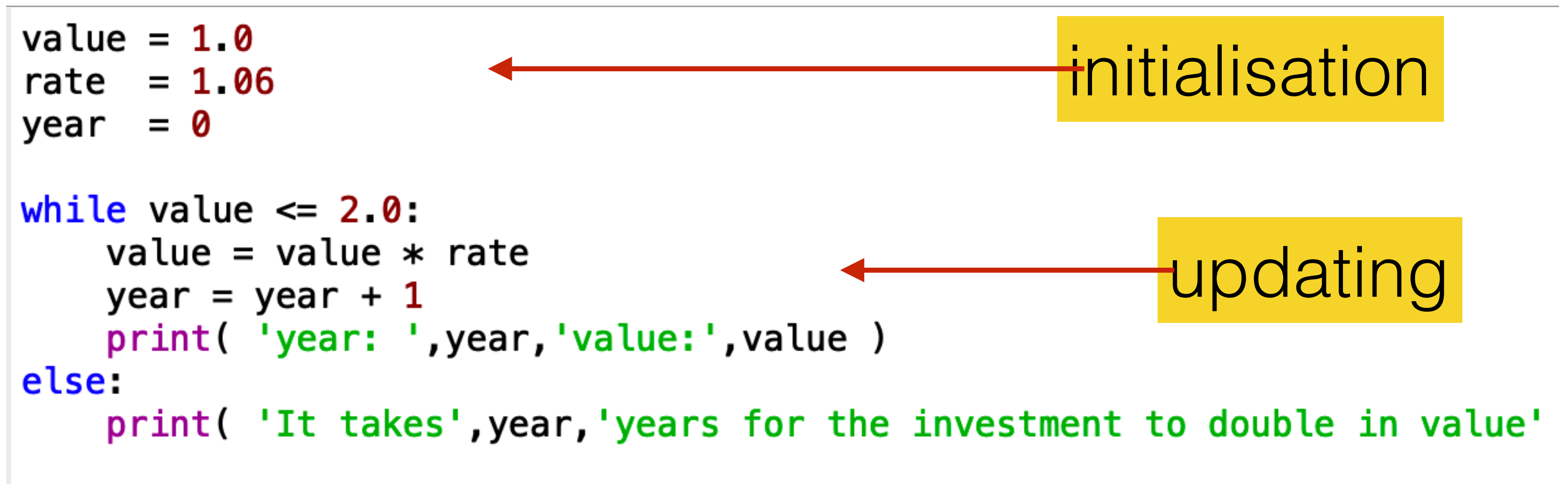
We will consider improved versions of this program

Extensions

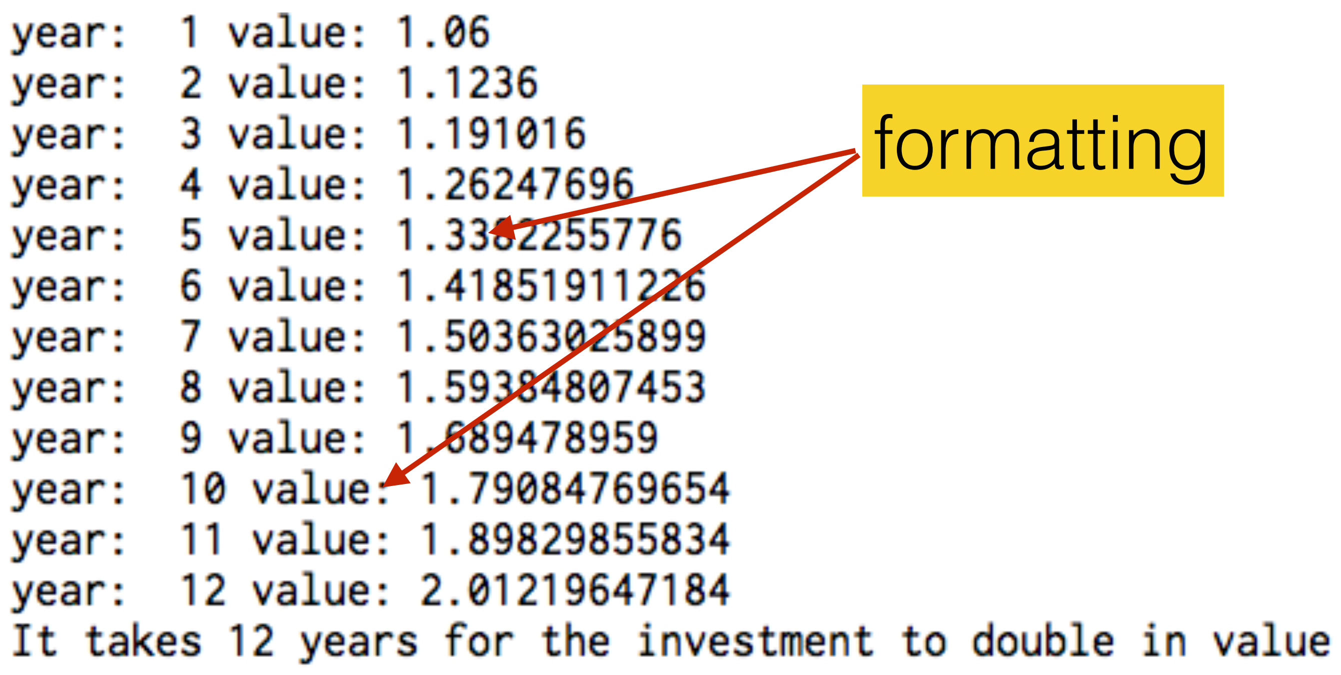
This is our first non-trivial example. It illustrates a few important Python topics:

1. Initialisation of variables
2. Updating variables
3. Formatted output

Python solution



```
value = 1.0
rate  = 1.06
year  = 0

while value <= 2.0:
    value = value * rate
    year = year + 1
    print( 'year: ',year,'value:',value )
else:
    print( 'It takes',year,'years for the investment to double in value'
```



```
year:  1 value: 1.06
year:  2 value: 1.1236
year:  3 value: 1.191016
year:  4 value: 1.26247696
year:  5 value: 1.3382255776
year:  6 value: 1.41851911226
year:  7 value: 1.50363025899
year:  8 value: 1.59384807453
year:  9 value: 1.689478959
year:  10 value: 1.79084769654
year:  11 value: 1.89829855834
year:  12 value: 2.01219647184
It takes 12 years for the investment to double in value
```

i) Initialisation

- It's good programming practice to define one's variables at the beginning of the program.
- In Python we need to initialise the variables by assigning a value (often, though not always, 0). This must be done *manually*.
- In Flowgorithm the situation is different:

  1. Declare variables.
  2. Assign values

These steps are combined in Python

Modification: undefined variable

```
value=1.0
rate=1.06

while value <= 2.0:
    value = value * rate
    year = year + 1
    print( 'year: ',year,'value:',value )
else:
    print( 'It takes',year,'years for the investment to double in value')
```

`year` isn't defined

```
Sec2.4-Loops/code/interest-undefined.py", line 6,
    year = year + 1

NameError: name 'year' is not defined
```

Python tells us where the problem first appears, but not where `year` should be defined

Modification: incorrect initial value

```
value = 1.0
rate = 2.06
year = 0

while value <= 2.0:
    value = value * rate
    year = year + 1
    print( 'year: ',year,'value:',value )
else:
    print( 'It takes',year,'years for the investment to double in value')
```

`rate` is defined incorrectly

```
year:  1 value: 2.06
It takes 1 years for the investment to double in value
```

This is a bug rather than a (syntax) error. While the program still runs, the output is incorrect.

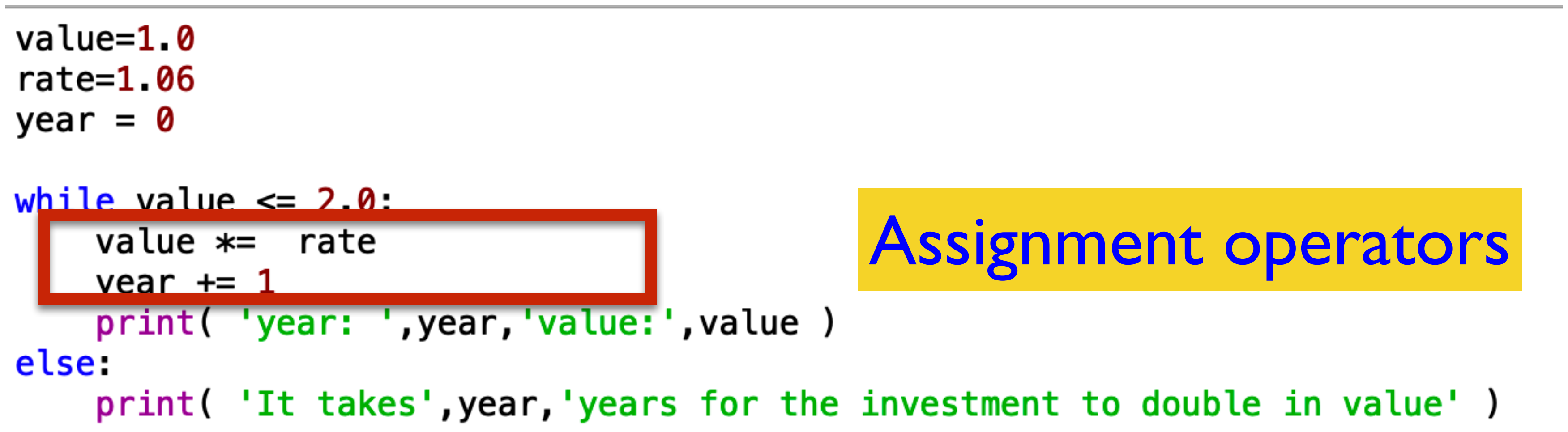
ii) Updating variables

- In science and engineering, many programs involve statements of the form `x = x +increment` or `x = x*scale`.
- This is sometimes referred to as an accumulator pattern.
- The interpretation is that we are updating a variable.
- *Examples:*
  - Position of an object as a function of time
  - Number of bacteria after a time interval

Assignment operators

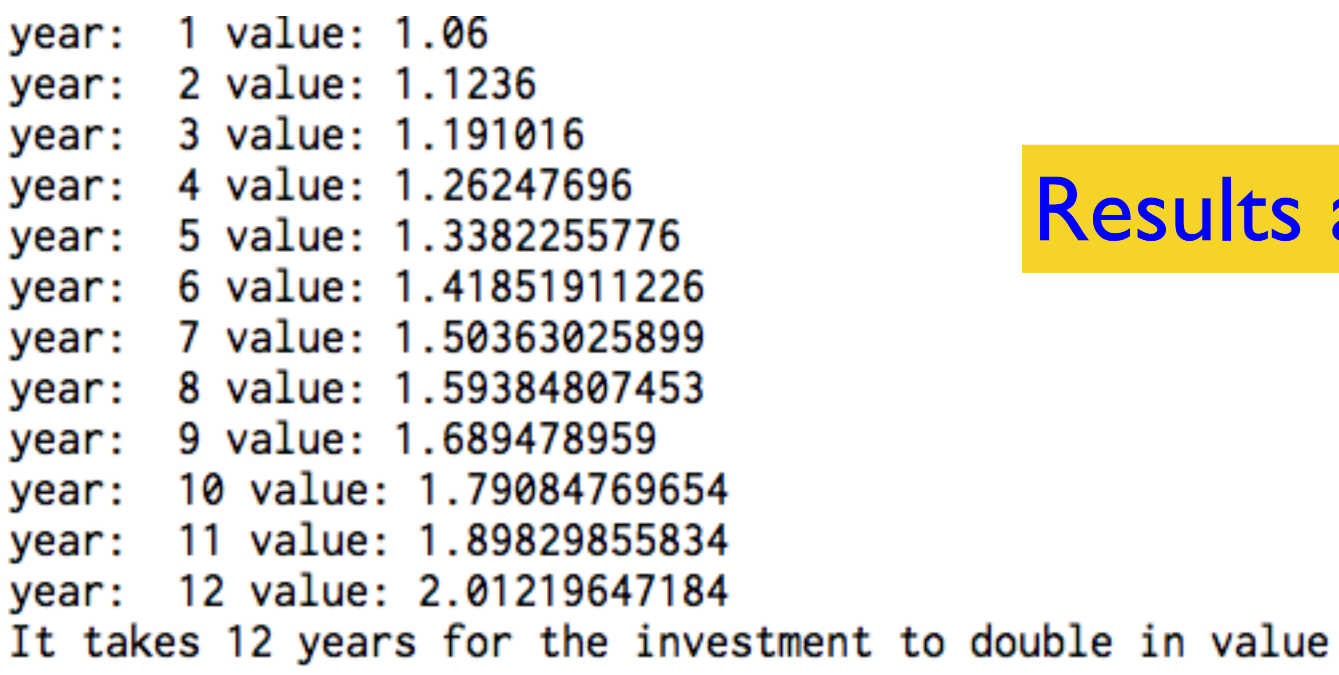
- We can abbreviate accumulator patterns using assignment operators:
    - `x = x + step → x += step`
    - `x = x - step → x -= step`
    - `x = x * constant → x *= constant`
    - `x = x / constant → x /= constant`
- As usual = refers to assignment not equality!

Modification: assignment operators



```
value=1.0
rate=1.06
year = 0

while value <= 2.0:
    value *=  rate
    year += 1
    print( 'year: ',year,'value:',value )
else:
    print( 'It takes',year,'years for the investment to double in value' )
```

Assignment operators



```
year:  1 value: 1.06
year:  2 value: 1.1236
year:  3 value: 1.191016
year:  4 value: 1.26247696
year:  5 value: 1.3382255776
year:  6 value: 1.41851911226
year:  7 value: 1.50363025899
year:  8 value: 1.59384807453
year:  9 value: 1.689478959
year:  10 value: 1.79084769654
year:  11 value: 1.89829855834
year:  12 value: 2.01219647184
It takes 12 years for the investment to double in value
```

Results are unchanged.

iii) Formatting

The output of our program is a little messy.

1. Python prints more digits than we need.
2. The output isn't nicely aligned.

We can correct these problems using formatted output.

Formatted output

- Up till now we've used this form of the `print` command:

```
print( string1,variable1,string2,variable,[…])
```

- Recall however the the general form of `print`:

```
print(<expression>)
```

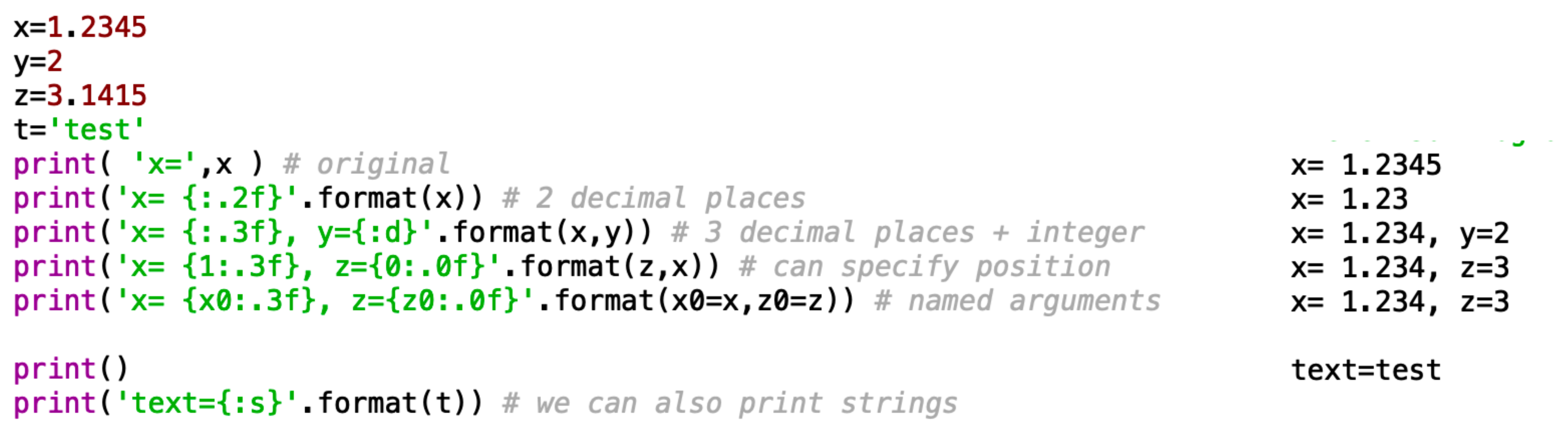
- We can choose an expression that allows us to specify the precision of the output. In its simplest form:

```
print('[text1] {spec}'.format(X))
```

where *`spec`* specifies the format string or how the variable `X` is to be formatted.

Format strings

There there are many ways of specifying the format string. The general syntax is complicated. For now, let's consider some typical examples.



```
x=1.2345
y=2
z=3.1415
t='test'
print( 'x=',x ) # original
print('x= {:.2f}'.format(x)) # 2 decimal places
print('x= {:.3f}, y={:d}'.format(x,y)) # 3 decimal places + integer
print('x= {1:.3f}, z={0:.0f}'.format(z,x)) # can specify position
print('x= {x0:.3f}, z={z0:.0f}'.format(x0=x,z0=z)) # named arguments

print()
print('text={:s}'.format(t)) # we can also print strings
```

```
x= 1.2345
x= 1.23
x= 1.234, y=2
x= 1.234, z=3
x= 1.234, z=3

text=test
```

Basic syntax

1. The value of a variable is substituted inside `{}`.
2. By default, the variables are assigned to the matching braces in the order in which they appear inside `.format()`.
3. Variables can be used in a different order by specifying the position with `{n}`where `n` is the order of the variable inside `.format()`.
4. The datatype can be specified as `{n:f}, {n:d}, {n:s}` for float, ints and strings.
5. The number of decimal places can be specified as `{n:.#f}` where # is the number of digits.

Modification: formatted output

```
value=1.0
rate=1.06
year = 0

while value <= 2.0:
    value *=  rate
    year += 1
    print( 'year: {:d}, value: {:.2f}'.format(year,value) )
else:
    print( 'It takes',year,'years for the investment to double in value' )
```

```
year: 1, value: 1.06
year: 2, value: 1.12
year: 3, value: 1.19
year: 4, value: 1.26
year: 5, value: 1.34
year: 6, value: 1.42
year: 7, value: 1.50
year: 8, value: 1.59
year: 9, value: 1.69
year: 10, value: 1.79
year: 11, value: 1.90
year: 12, value: 2.01
It takes 12 years for the investment to double in value
```

The output looks nicer! But it's not perfect

Columns aren't aligned

Alignment

There are various ways of aligning the text.

1. Specify the number of digits of output as `{n:#d}` where `#` is the number of digits. This ensures that the output has a fixed number of digits.
2. Add a tab character with `\t`. This is equivalent to using a tab stop in a word processor like Word.

Modification: padding

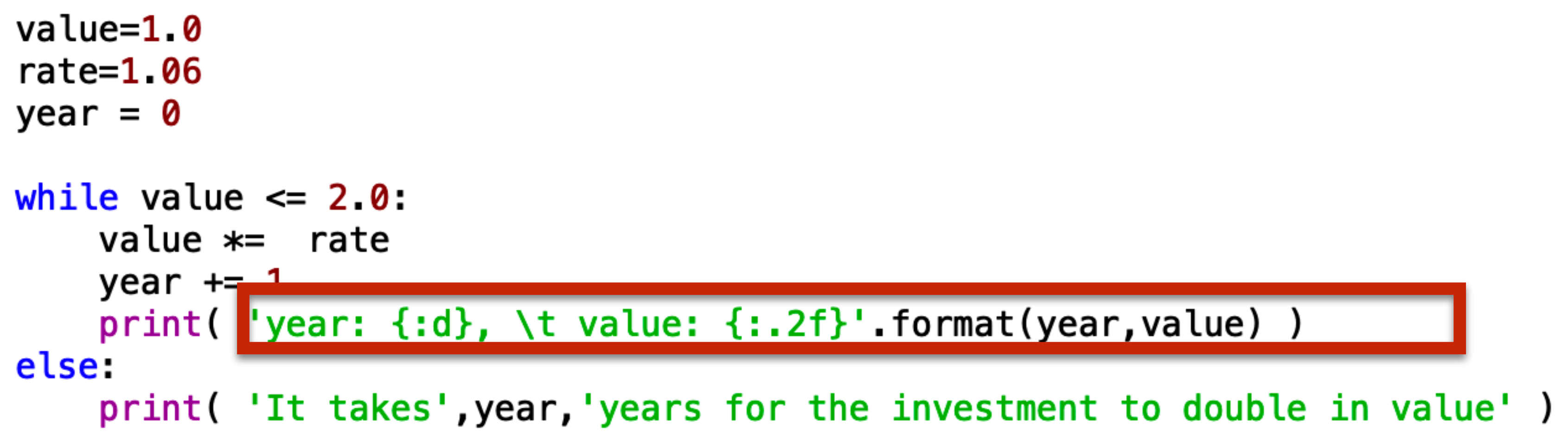
```
value=1.0
rate=1.06
year = 0

while value <= 2.0:
    value *=  rate
    year += 1
    print( 'year: {:3d}, value: {:.2f}'.format(year,value) )
else:
    print( 'It takes',year,'years for the investment to double in value' )
```

```
year:   1, value: 1.06
year:   2, value: 1.12
year:   3, value: 1.19
year:   4, value: 1.26
year:   5, value: 1.34
year:   6, value: 1.42
year:   7, value: 1.50
year:   8, value: 1.59
year:   9, value: 1.69
year:  10, value: 1.79
year:  11, value: 1.90
year:  12, value: 2.01
It takes 12 years for the investment to double in value
```

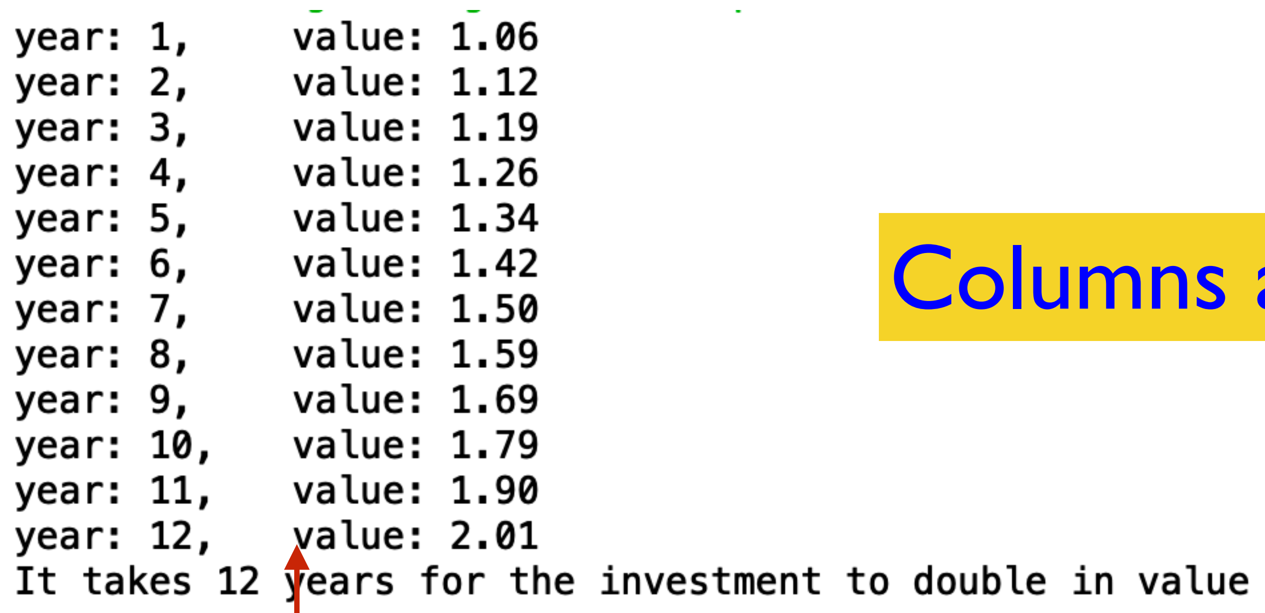
Columns are now aligned!

Output is padded with extra blank spaces.

Modification: tab



```
value=1.0
rate=1.06
year = 0

while value <= 2.0:
    value *=  rate
    year += 1
    print( 'year: {:d}, \t value: {:.2f}'.format(year,value) )
else:
    print( 'It takes',year,'years for the investment to double in value' )
```



```
year: 1,	value: 1.06
year: 2,	value: 1.12
year: 3,	value: 1.19
year: 4,	value: 1.26
year: 5,	value: 1.34
year: 6,	value: 1.42
year: 7,	value: 1.50
year: 8,	value: 1.59
year: 9,	value: 1.69
year: 10,	value: 1.79
year: 11,	value: 1.90
year: 12,	value: 2.01
It takes 12 years for the investment to double in value
```

Columns are now aligned!

Output is shifted to
the next ‘tab stop’.

3. `for` loop

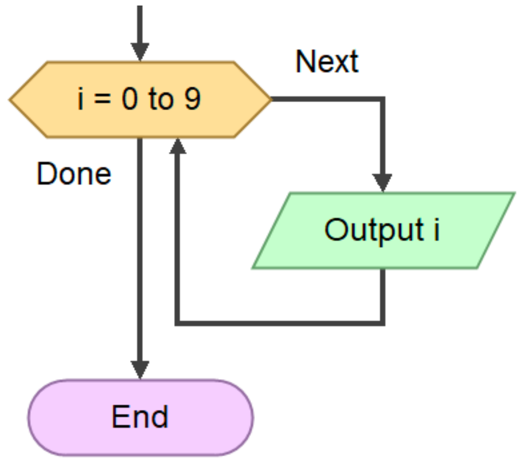
Motivation

- Another way of repeating a task is to do it for a specified number of times.
- *Examples*
  - Add 4 cups of flour
  - Buy 4 large apples
  - Send 4 text messages

Note: this is very similar to the while loop over an index. However, instead of specifying a loop condition (e.g., i<10), we specify the range of numbers.

Overview

The `for` loop also executes a series of statements while a condition is true. But it's simpler to associate the loop with certain iterations over values of the index variable.



```
for i in range(1,11):
    print(i)
```

Flowgorithm

Python

Simplest form of the `for` loop

Most of the time we'll use `for` with `range`:

```
for variable in range([start,]stop[,step]):
    statement 1.1
    statement 1.2
    [...]
[...]
```

executed *for each number in the range*

executed *after loop has completed*

Syntax follows that of `while` and `if`

`range`

`range` effectively generates an integer **list** with a specified starting point, end point and spacing. Syntax:

```
range([start,]stop[,step])
```

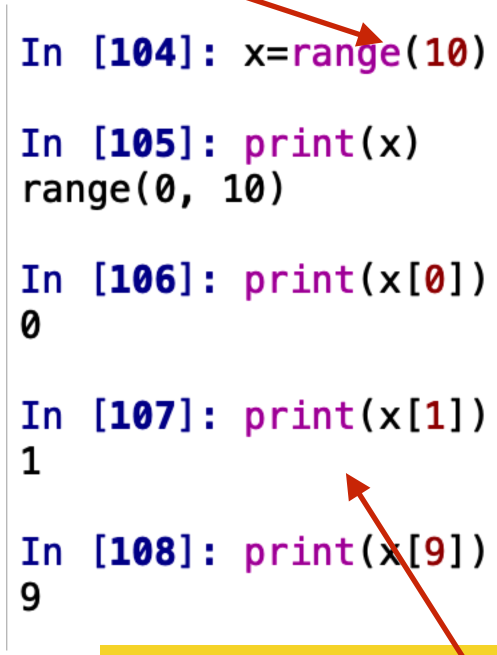
where

- `start` is the optional starting point (default =0)
- `stop` is the end point (loop terminates at `stop`-1 for increment >0)
- `step` is the optional spacing (default =1)

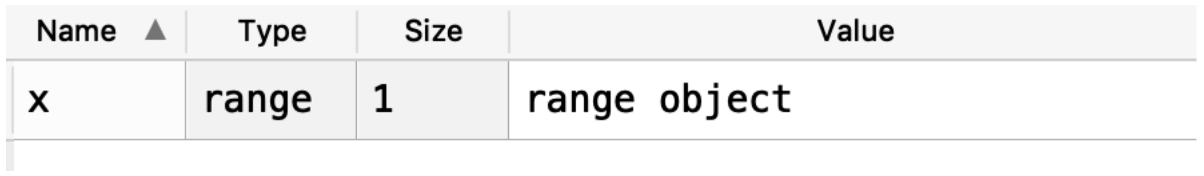
Comments

In Python 2, `range` generated an actual list. In Python 3, it generates an object with a special range data type.

equivalent to [0,1,2,3,4,5,6,7,8,9]



```
In [104]: x=range(10)

In [105]: print(x)
range(0, 10)

In [106]: print(x[0])
0

In [107]: print(x[1])
1

In [108]: print(x[9])
9
```



Name ▲	Type	Size	Value
x	range	1	range object

can access individual elements

Simple examples

```
for i in range(10):
    print(i)
```

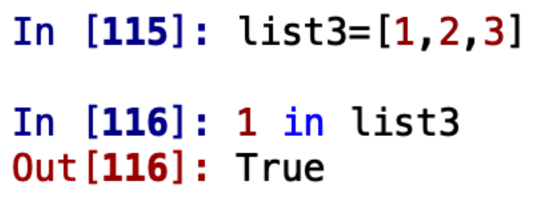
```
for i in range(10,101,10):
    print('The sqrt of {0:3d} = {1:.2f}'.format(i,float(i)**0.5))
```

```
0
1
2
3
4
5
6
7
8
9
```

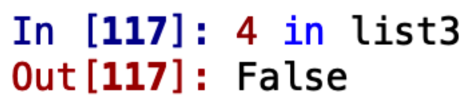
```
The sqrt of  10 = 3.16
The sqrt of  20 = 4.47
The sqrt of  30 = 5.48
The sqrt of  40 = 6.32
The sqrt of  50 = 7.07
The sqrt of  60 = 7.75
The sqrt of  70 = 8.37
The sqrt of  80 = 8.94
The sqrt of  90 = 9.49
The sqrt of 100 = 10.00
```

Reminder about `in`

We have already introduced `in!` Recall that it's a way of testing for membership in a list.



```
In [115]: list3=[1,2,3]

In [116]: 1 in list3
Out[116]: True
```



```
In [117]: 4 in list3
Out[117]: False
```

Thus the `for` loop

```
for variable in range([start,]stop[,step]):
```

is executed for all variables belonging to the `range` list.

General syntax of the `for` statement

More precisely, the `for` statement has the following syntax:

```
for variable in list:
    statement 1.1
    statement 1.2
    [...]
statement 2.1
statement 2.1
[...]
```

executed for each element in the list

executed *after* every item in the list has been examined

Comments

- For the simplest version of the `for` loop, the loop variable is an index or counter variable. In this case, it's almost always an integer.
- It's possible to modify the counter inside the loop, but this can lead to confusion.

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

```
for i in range(10):
    i = 1
    print(i)
```

Modification has no effect on loop

```
1
1
1
1
1
1
1
1
1
1
```

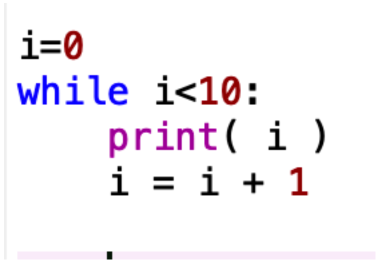
The program still runs but it can be hard to understand what's happening

Why is the `for` loop useful?

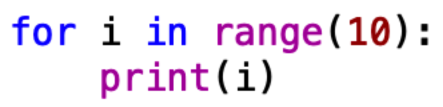
- The `for` loop is probably the most commonly used loop in science and engineering.
- Why? In many applications, a set of tasks is repeated a certain number of times.
  - *Examples*: set of molecules or spatial locations.
- More concretely, many applications in science and engineering are related to counting.

Differences between `while` and `for` (1)

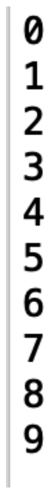
The `while` and `for` loops are very similar. Everything we can do with `for` we can do with `while`.



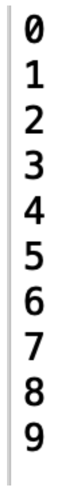
```
i=0
while i<10:
    print( i )
    i = i + 1
```



```
for i in range(10):
    print(i)
```



```
0
1
2
3
4
5
6
7
8
9
```

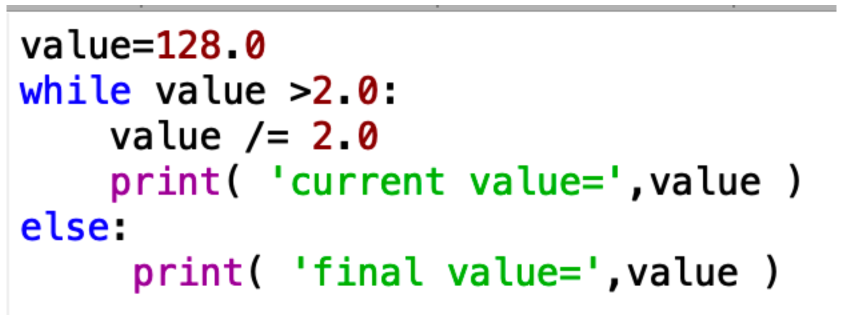


```
0
1
2
3
4
5
6
7
8
9
```

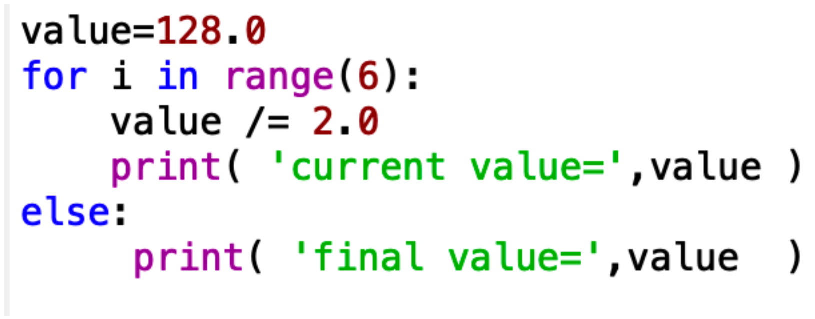
The `for` loop is more compact.

Differences between `while` and `for` (2)

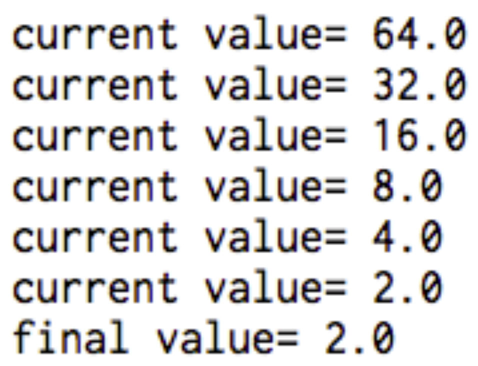
The `while` loop is more useful when we don't know how many times the loop will be executed (e.g. finite loop):



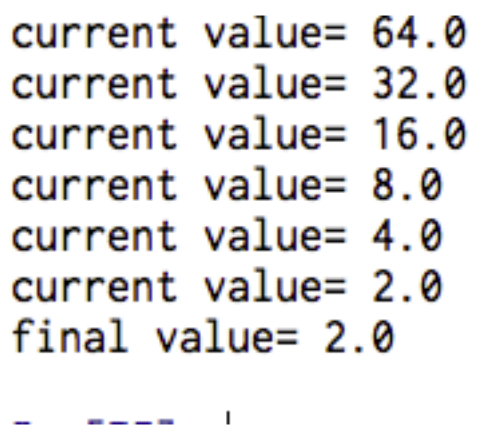
```
value=128.0
while value >2.0:
    value /= 2.0
    print( 'current value=',value )
else:
     print( 'final value=',value )
```



```
value=128.0
for i in range(6):
    value /= 2.0
    print( 'current value=',value )
else:
     print( 'final value=',value  )
```



```
current value= 64.0
current value= 32.0
current value= 16.0
current value= 8.0
current value= 4.0
current value= 2.0
final value= 2.0
```



```
current value= 64.0
current value= 32.0
current value= 16.0
current value= 8.0
current value= 4.0
current value= 2.0
final value= 2.0
```

The version with the `for` loop isn't as useful in practice.

Use of `else`

- In theory we can add an `else` block after a `for` block.
- In practice, there's usually no need because the statements after the `for` will be automatically executed.

```
for i in range(3):
    print( i )
print( "we're done" )
```

```
0
1
2
we're done
```

```
for i in range(3):
    print(i)
else:
    print( "we're done" )
```

```
0
1
2
we're done
```

```
for i in range(3):
    print( i )
else:
    print( "we're done" )
print( "we're done" )
```

```
0
1
2
we're done
we're done
```

Iteration

- In traditional computer languages, `for` loops are limited to counting, e.g., `for` with an appropriately specified `range`.
- This is limiting because it isn't always convenient to think in terms of a counter variable.
- Sometimes it's more convenient to loop over items of a list. This is referred to as iteration rather than counting. Iteration is more general. In Python this is easily done using `in`.

Looping over a counter variable vs. iteration over a list

For some problems it's more natural to iterate over a list.

```
mylist= ['dog', 'cat', 'mouse', 'goldfish', 'turtle']
N=len(mylist)
for i in range(N):
    print( mylist[i] )
```

counter variable

```
mylist= ['dog', 'cat', 'mouse', 'goldfish', 'turtle']
for animal in mylist:
    print( animal )
```

iteration over list

```
dog
cat
mouse
goldfish
turtle
```

Iteration yields more compact code as there's no need to deal with a counter variable.

Extensions

It's easy to think of situations in which the basic `for` loop is inadequate.

Examples

- Want a counter variable when we iterate over a list
- Want to loop over multiple variables
- Want to stop repeating the task earlier than expected

We now consider several extensions.

i) Extracting a counter variable

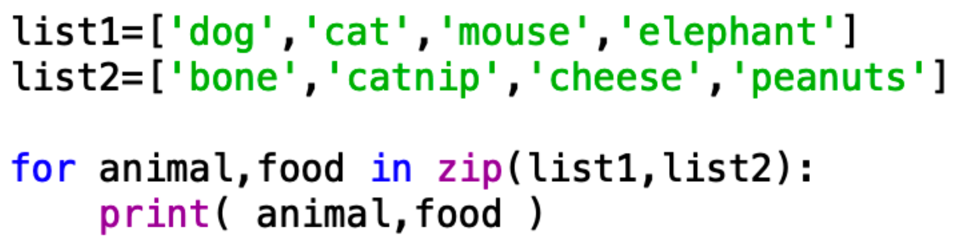
On occasion we want to iterate over a list but still have access to a counter variable. This can be done using `enumerate`.

```
list1=['dog','cat','mouse']
list2=['bone','catnip','cheese']

for i,animal in enumerate(list1):
    print( i,animal,list2[i] )
```

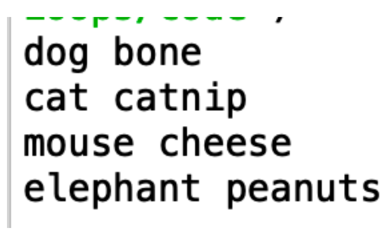
```
0 dog bone
1 cat catnip
2 mouse cheese
```

ii) Iterating over multiple lists

Occasionally we can to iterate over multiple lists. This can be using `zip`.



```
list1=['dog','cat','mouse','elephant']
list2=['bone','catnip','cheese','peanuts']

for animal,food in zip(list1,list2):
    print( animal,food )
```



```
dog bone
cat catnip
mouse cheese
elephant peanuts
```

iii) Nested loops

Just as with `if`, `for` loops can be nested arbitrarily many times.

```
for i in range(9):
    print ('i=',i)
```

```
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
```

```
for i in range(3):
    for j in range(3):
        print( 'i=',i,'j=',j )
```

```
i= 0 j= 0
i= 0 j= 1
i= 0 j= 2
i= 1 j= 0
i= 1 j= 1
i= 1 j= 2
i= 2 j= 0
i= 2 j= 1
i= 2 j= 2
```

Why are nested loops useful?

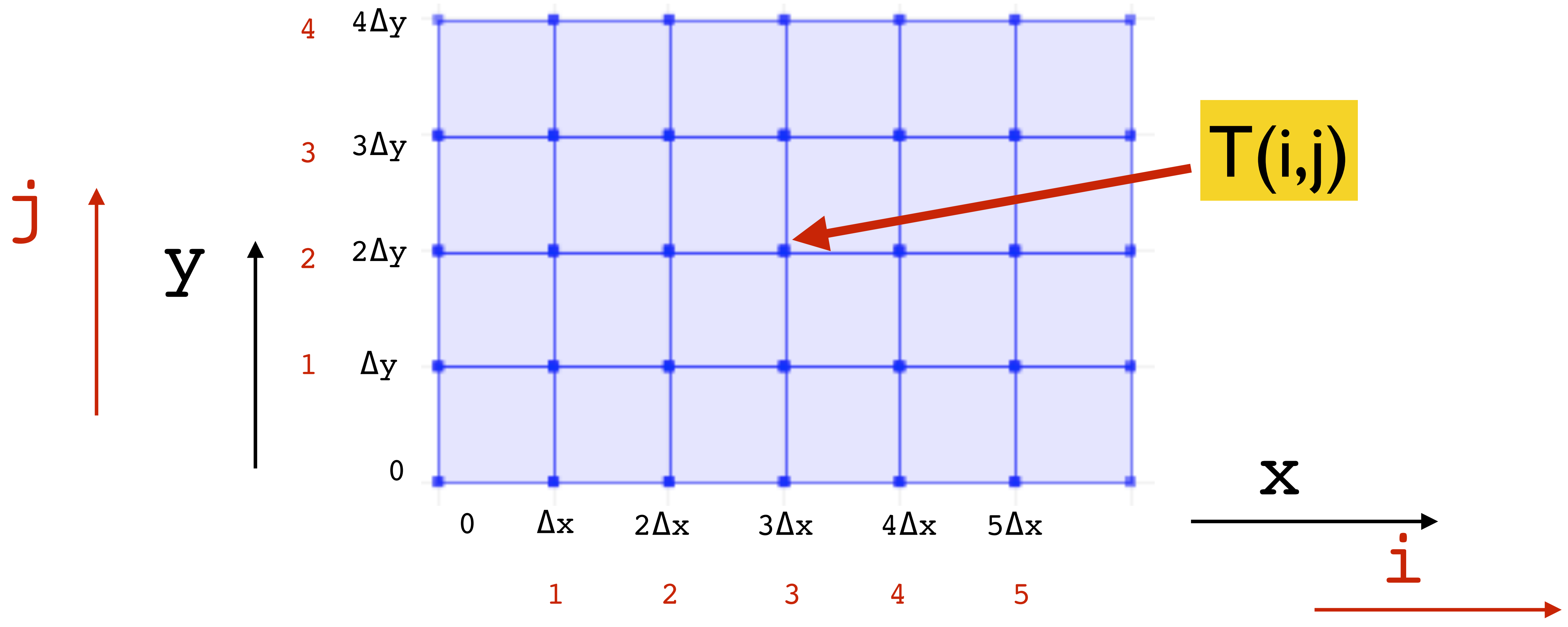
In many programs, each loop corresponds to a distinct direction. *Examples:*

- `row, column`
- `x,y`
- `x,y,z`

The nested loop allows us to sweep through all the points in the domain in a very natural way.

Application: defining a 2-D field

We can also define a field on our grid. This can be done by looping over `(i,j).`

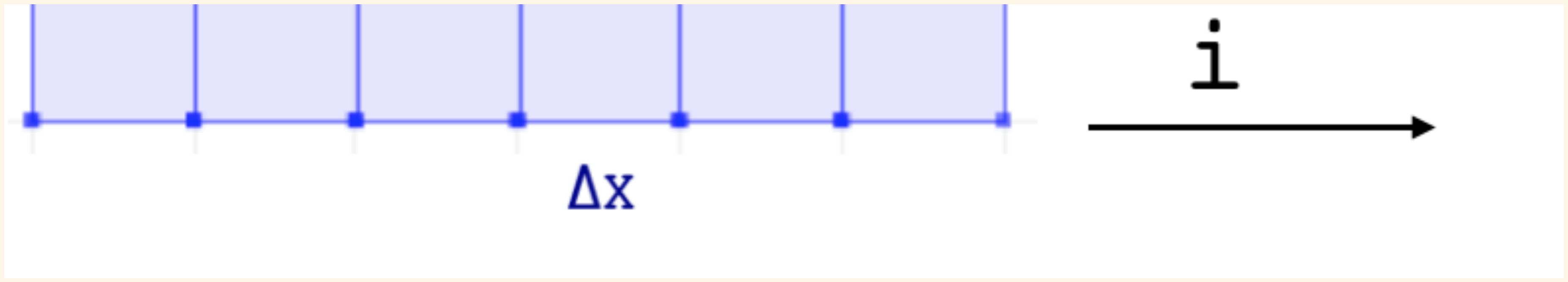


Grid spacing

- In the previous figure `i` and `j` are integers. In reality they correspond to locations `x` and `y`.
- In many situations we would like to convert (`i,j`) coordinates to (`x,y`) coordinates. This is easy to do!
  - `i=0 → x=0`
  - `i=1 → x=Δx`   Δx is the grid spacing
  - `i=2 → x=2Δx`
  - `etc.`
- To have `N` points between x_1 and x_N, use $\Delta x = (x_N - x_1)/(N-1)$. `N` includes endpoints.

Example 4: a 1-D grid

Consider a one-dimensional grid with 20 evenly spaced points and endpoints at `xi=0` and `xf=10`. Use a `for` loop to print the position of each grid point `xi`.



iv) Speeding up execution of a loop

- By design, the same instructions are executed for every single pass through the loop. This can take a long time if there are many instructions to be carried out or many steps in the loop.
- There are two ways to speed things up:

1. Jump out of the loop (*break*)
2. Proceed to the next iteration of the loop (*continue*)

`break`

- `break` causes Python to jump from the current loop to the (adjacent) outer loop. *This amounts to being kicked out of the current/inner loop.*
- It also applies to `while`.

current loop

```
for i in range(10):
    if i==2:
        break
    else:
        print( i )
print( 'we are finished' )
```

```
i= 0
i= 1
we are finished
```

break at i=2

current loop

```
for j in range(3):
    print()

    for i in range(10):
        if i==3:
            break
        else:
            print('j=',j,'i=',i)

print( 'bye' )
```

```
j= 0 i= 0
j= 0 i= 1
j= 0 i= 2

j= 1 i= 0
j= 1 i= 1
j= 1 i= 2

j= 2 i= 0
j= 2 i= 1
j= 2 i= 2
bye
```

break at i=3

continue

`continue` causes Python to immediately proceed with the next iteration of the current loop. *This amounts to skipping an iteration.*

current loop

```
for i in range(5):
    if i==3:
        continue
    else:
        print('i=', i)
print('bye')
```

```
i= 0
i= 1
i= 2
i= 4
bye
```

skip i=3

current loop

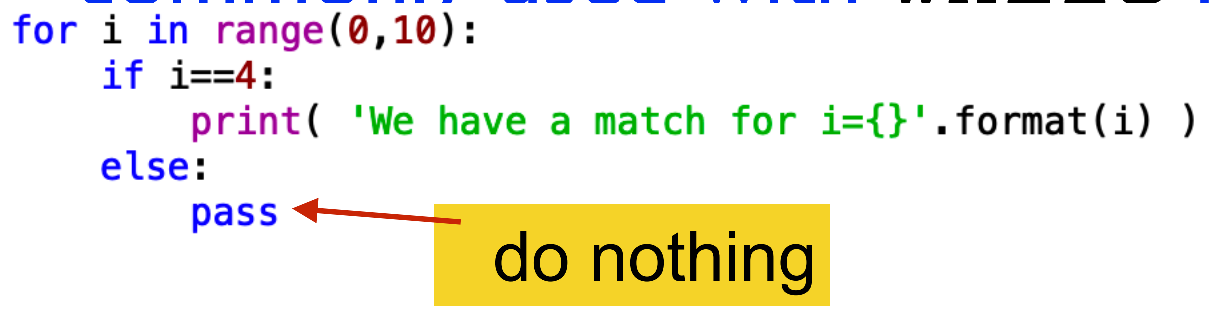
```
for j in range (3):
    print()
    for i in range(3):
        if i==1:
            continue
        else:
            print('j=',j,'i=',i)
```

```
j= 0 i= 0
j= 0 i= 2

j= 1 i= 0
j= 1 i= 2

j= 2 i= 0
j= 2 i= 2
```

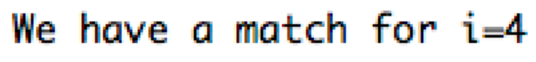
skip i=1

pass

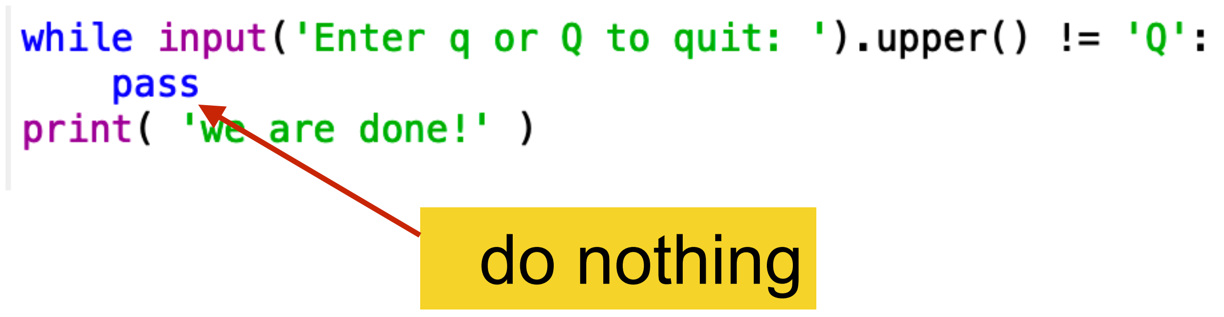
- `pass` causes Python to do nothing!
- It's effectively the same as a comment. However, a comment may generate a syntax error if there are no other statements. It's equivalent to continue if we have no other statements.
- It can also be used with `while`. In fact, it's more commonly used with `while` rather than `for`.



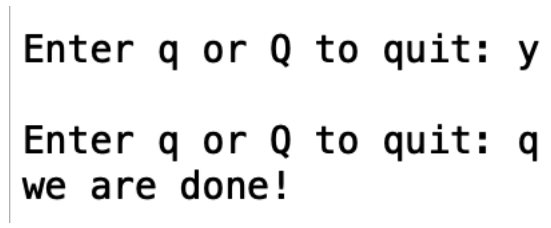
```
for i in range(0,10):
    if i==4:
        print( 'We have a match for i={}'.format(i) )
    else:
        pass
```



```
We have a match for i=4
```



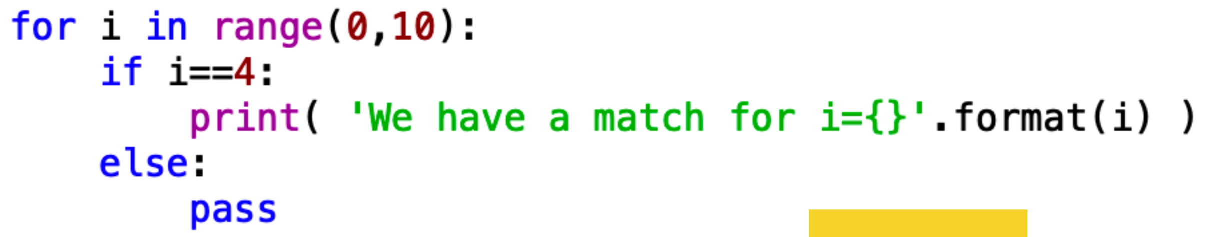
```
while input('Enter q or Q to quit: ').upper() != 'Q':
    pass
print( 'we are done!' )
```



```
Enter q or Q to quit: y

Enter q or Q to quit: q
we are done!
```

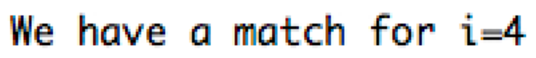
Difference between `pass` and `continue`

- In practice, we can use `continue` instead of `pass`.
- However, `pass` is more suggestive of doing nothing.

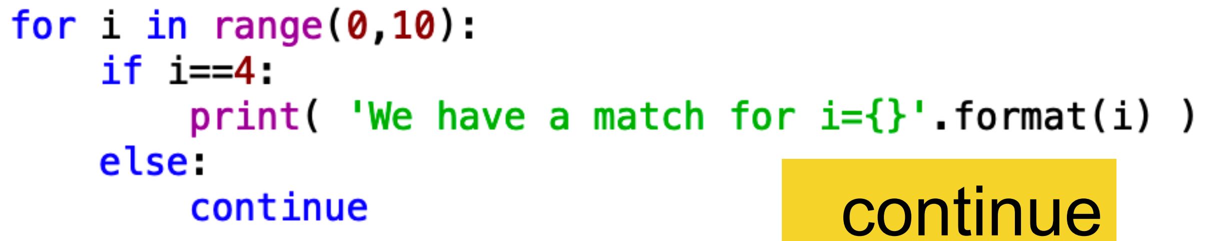


```
for i in range(0,10):
    if i==4:
        print( 'We have a match for i={}'.format(i) )
    else:
        pass
```

pass

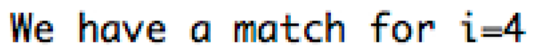


```
We have a match for i=4
```



```
for i in range(0,10):
    if i==4:
        print( 'We have a match for i={}'.format(i) )
    else:
        continue
```

continue



```
We have a match for i=4
```

Summary

1. Loops are fundamental to computer programming.
2. `while` loops are useful for testing whether a condition has changed.
3. `for` loops are useful for counting or iterating over the items of a list.