Introduction to Python 2

August 2–3, 2023

1pm–3:30pm CDT

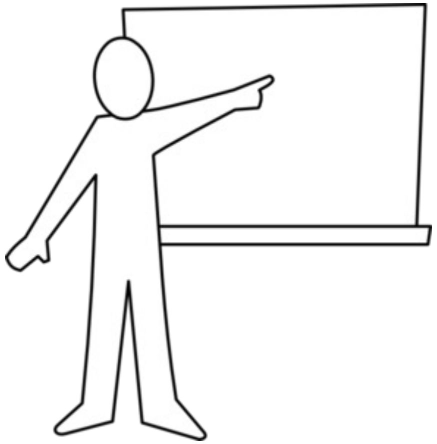
Hello, world!

Dr. Amanda Kis ("Amanda")

π $\approx$

$\div$

$\sum$



Documentation & tutorials

- Many of the definitions used in this workshop are taken from the official Python documentation on Python.org
  - Check out their tutorial for a comprehensive but informal overview

- Other great tutorials, courses, and overviews
  - LearnPython.org interactive python tutorial
  - freeCodeCamp.org Learn Python full course (YouTube video)
  - Real Python tutorials (my favorite!)
  - Codeacademy Learn Python 2 course

Google Colaboratory (“Google Colab”)

- https://colab.research.google.com/

- A Google Account (e.g., a gmail account or another email account you have connected to Google) is required

- File → Upload notebook
  - Upload the .ipynb file sent for this workshop via the Upload tab
  - Or if you saved the .ipynb file to Google Drive, find and click on it from there

Sequences

- Groups of objects that are ordered based on their position
- Capable of *iteration*: returning their items one at a time
- Strings and lists are examples of sequences in Python

```
motto = ‘Cats rock!’ # string

my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’] # list
```

Lists

- Group of comma-separated items, surrounded by square brackets
- Items usually are *homogeneous* (of same type)

```
>>> my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’]
```

All items are strings

- Items can be *nonhomogeous* (of different types)

```
>>> different_list = [10, ‘Oklahoma’, -5.6, True]
```

Items are a mix of integer, string, float, and Boolean types

- Lists can be empty

```
>>> empty_list = []
```

Indexing

- Access an individual item of a sequence based on its position
- *Zero-based*: First (left-most) item has index 0, indices increment by 1 to the right to the end of the sequence

```
         0123456789
motto = ‘Cats rock!’
```

```
           0       1     2      3         4
my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’]
```

Indexing

- Access an individual item of a sequence based on its position
- *Zero-based*: First (left-most) item has index 0, indices increment by 1 to the right to the end of the sequence

```
             0123456789
>>> motto = ‘Cats rock!’

>>> motto[0]
‘C’
>>> motto[5]
‘r’
```

Indexing

- Access an individual item of a sequence based on its position
- *Zero-based*: First (left-most) item has index 0, indices increment by 1 to the right to the end of the sequence

```
                 0      1     2      3         4
>>> my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’]

>>> my_list[0]
‘This’
>>> my_list[3]
‘Python’
```

Indexing

- The largest index, for the last (right-most) item, is one less than the length of the sequence

```
             0123456789
>>> motto = ‘Cats rock!’

>>> len(motto)
10
>>> motto[9]
‘!’
>>> motto[10]
IndexError: string index out of range
```

```
                 0     1     2      3         4
>>> my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’]

>>> len(my_list)
5
>>> my_list[4]
‘Python’
>>> my_list[5]
IndexError: list index out of range
```

Indexing

- The largest index, for the last (right-most) item, is one less than the length of the sequence

```
             0123456789
>>> motto = ‘Cats rock!’

>>> len(motto)
10
>>> motto[9]
‘!’
>>> motto[10]
IndexError: string index out of range
```

len() function

built-in function that returns the number of items in an iterable object

```
                 0      1     2      3         4
>>> my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’]

>>> len(my_list)
5
>>> my_list[4]
‘Python’
>>> my_list[5]
IndexError: list index out of range
```

Indexing

- Negative indices start at -1 for the right-most item and increment by -1 to the left to the start of the sequence

```
             -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
>>> motto = ‘Cats rock!’

>>> motto[-1]
‘!’
>>> motto[-10]
‘C’
>>> motto[-11]
IndexError: string index out of range
```

```
                  -5       -4    -3     -2         -1
>>> my_list = [‘This’, ‘is’, ‘a’, ‘Python’, ‘class’]

>>> my_list[-1]
‘class’
>>> my_list[-5]
‘This’
>>> my_list[-6]
IndexError: list index out of range
```

Slicing

- Access a subsequence (range of items) in a sequence
- Builds on indexing: [start:stop:step]
  - start is included in the subsequence
  - stop is excluded from the subsequence
  - step gives the increment between each index

```
             0123456789
>>> motto = ‘Cats rock!’

>>> motto[1:6:1]
‘ats r’
>>> motto[3:9:2]
‘src’
```

Slicing

- Access a subsequence (range of items) in a sequence
- Builds on indexing: [start:stop:step]
    - start is included in the subsequence
    - stop is excluded from the subsequence
    - step gives the increment between each index

} **half-open interval [start, stop)**

```
             0 1 2 3 4 5 6 7 8 9
>>> motto = ‘Cats rock!’

>>> motto[1:6:1]
‘ats r’
>>> motto[3:9:2]
‘src’
```

Slicing

- [start:stop:step]
  - If start is omitted, the subsequence begins with the first item of the sequence
  - If stop is omitted, the subsequence proceeds through the last item of the sequence
  - If step is omitted, an increment of 1 is used

```
                0 1 2 3 4 5 6 7 8 9
>>> motto = ‘Cats rock!’

>>> motto[:4] # start from index 0 with step = 1
‘Cats’
>>> motto[5:] # start from index 5, proceed to end of sequence, with step = 1
‘rock!’
>>> motto[5::2] # start from index 5, proceed to end of sequence, with step = 2
‘rc!’
>>> motto[::2] # start from index 0, proceed to end of sequence every other item is returned
‘Ct ok’
```

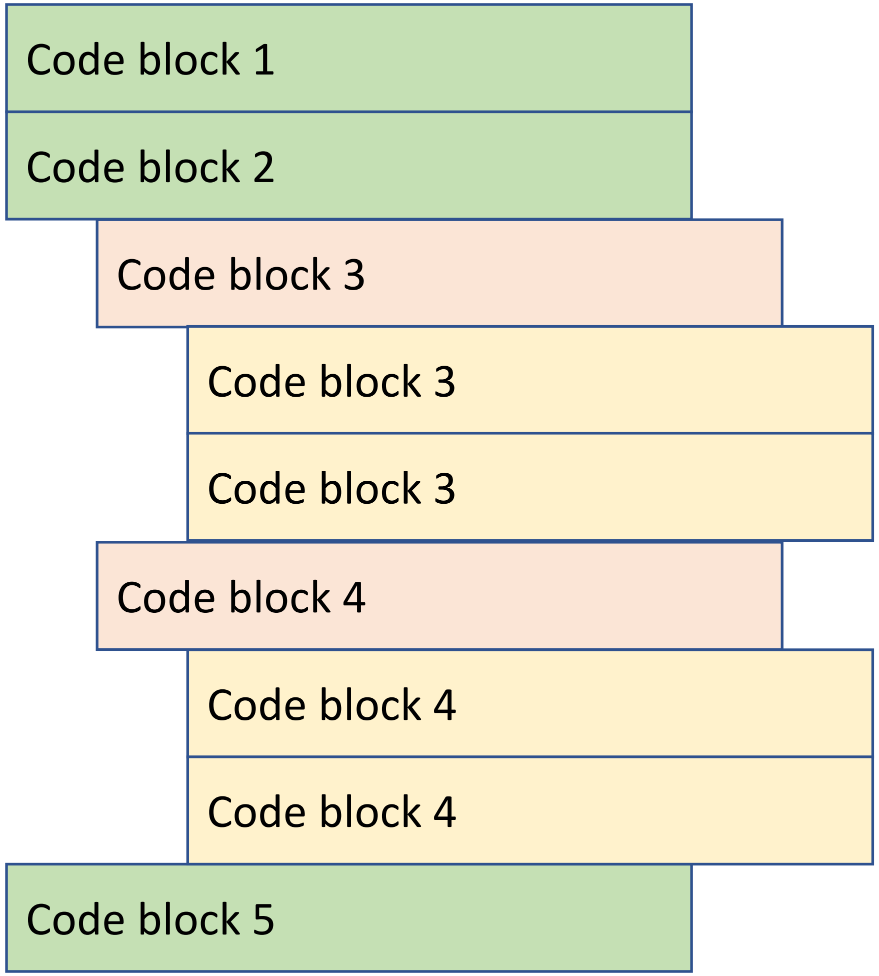
Control flow

- The order in which pieces of a computer program are executed

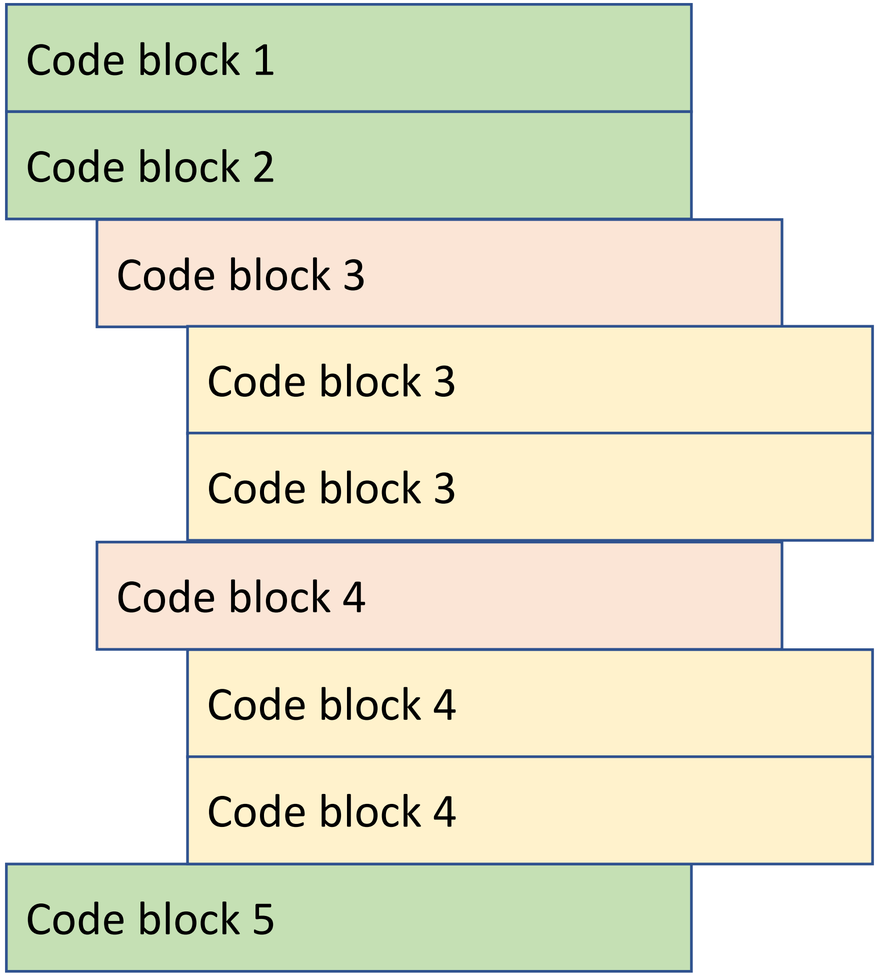
- Control flow in Python is determined by:
    - Indentation
    - Conditional execution
        - `if` statements
    - Loops
        - `for` loops
        - `while` loops
    - Functions

Indentation

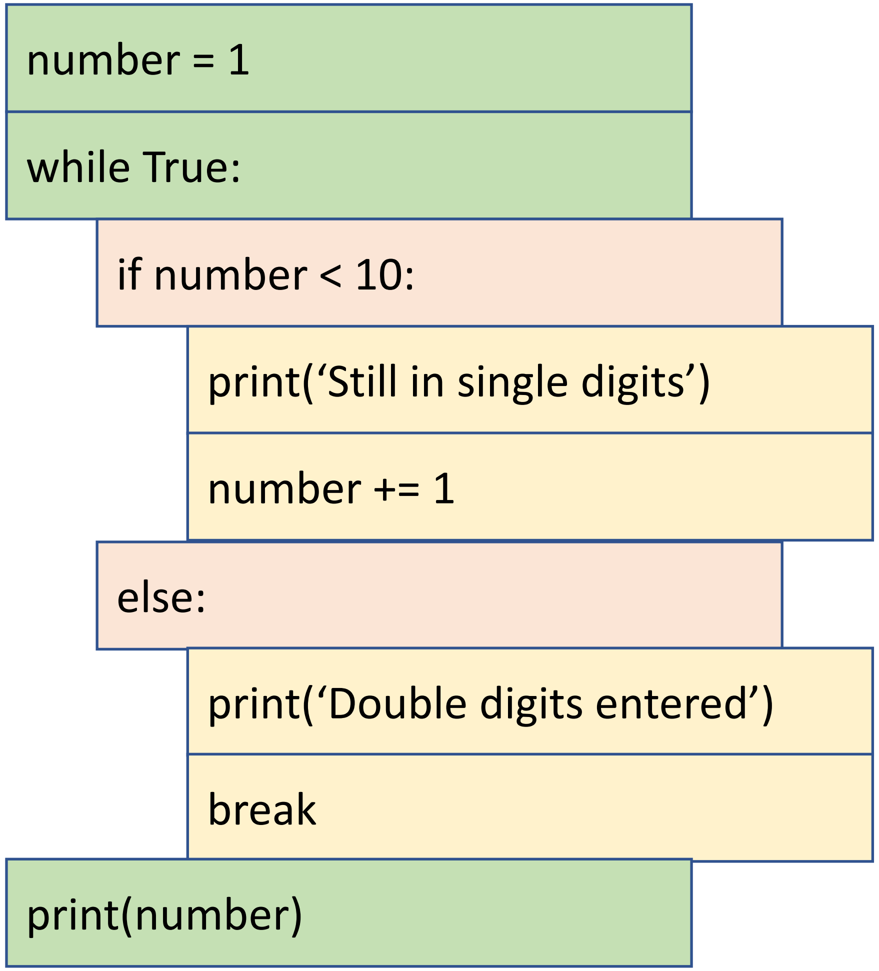
- Leading whitespace produced by either spaces or tabs (use one or the other—do not mix!)
- Used to group statements
- Code at smaller indentation level given higher priority
- *Code blocks*: Pieces of code executed as a unit
- Indentation defines multi-line code blocks
    - Indentation starts a multi-line code block
    - Dedentation ends a multi-line code block



Indentation

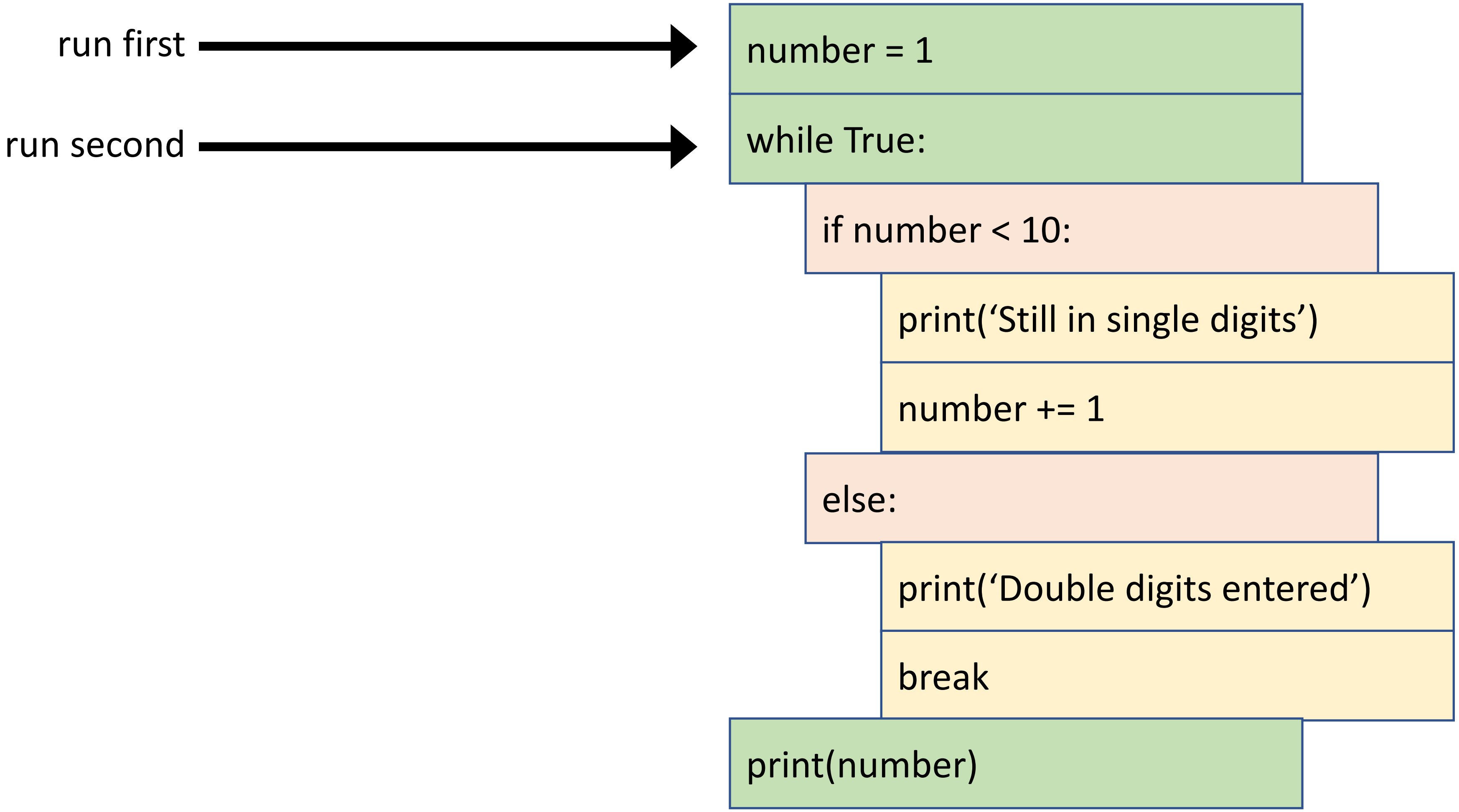


```
Code block 1
Code block 2
    Code block 3
        Code block 3
        Code block 3
    Code block 4
        Code block 4
        Code block 4
Code block 5
```



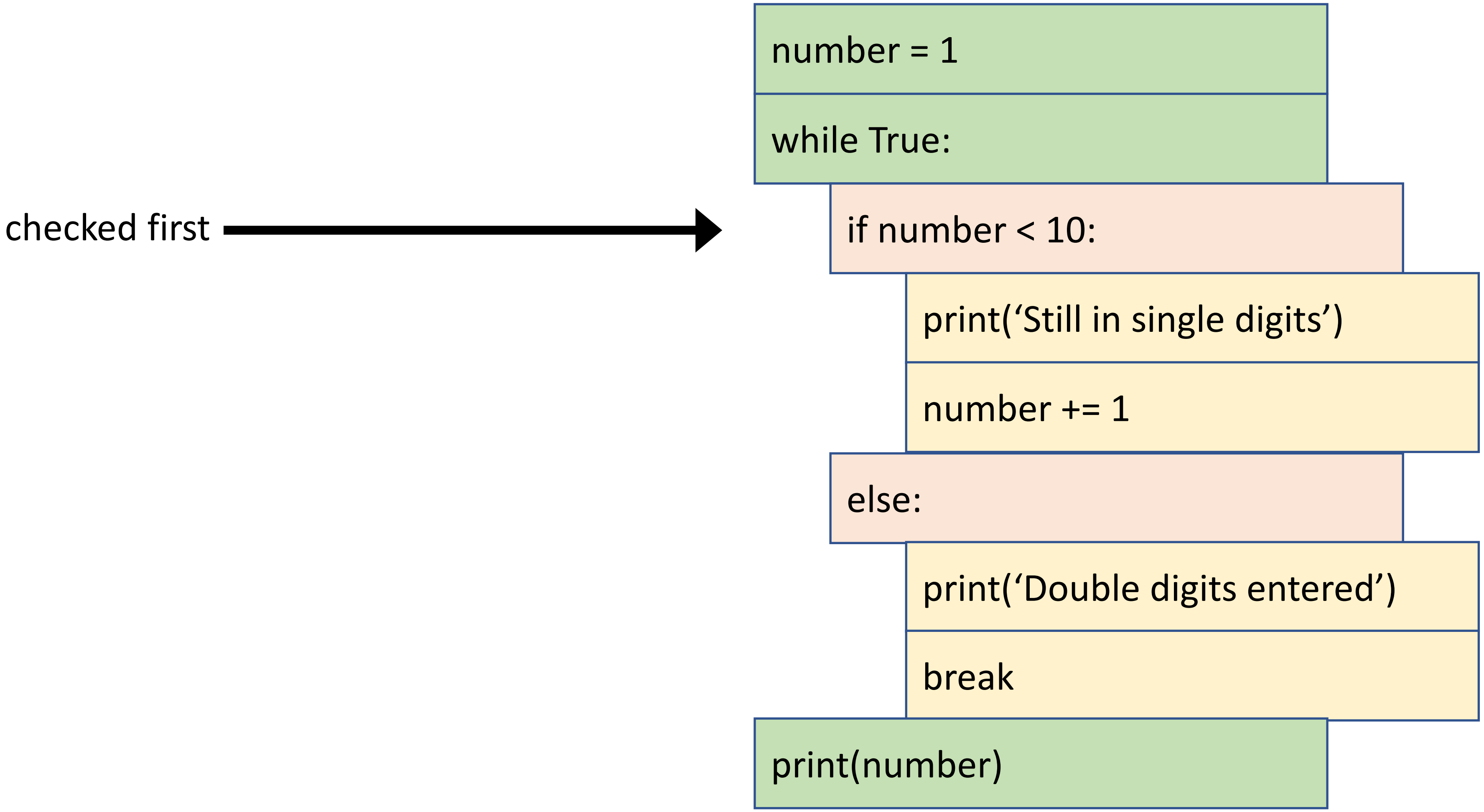
```
number = 1
while True:
    if number < 10:
        print(‘Still in single digits’)
        number += 1
    else:
        print(‘Double digits entered’)
        break
print(number)
```

Indentation



```
number = 1
while True:
    if number < 10:
        print(‘Still in single digits’)
        number += 1
    else:
        print(‘Double digits entered’)
        break
print(number)
```

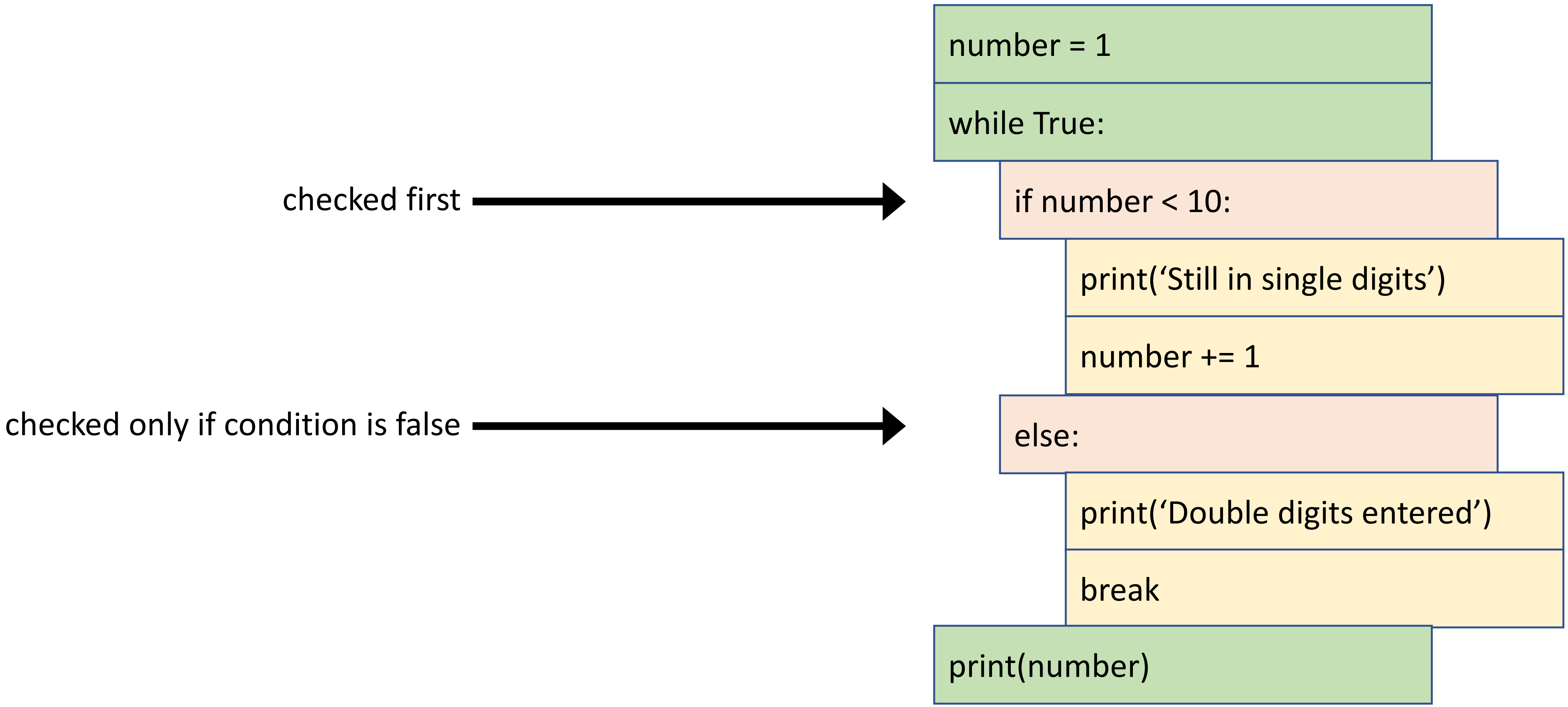
Indentation



checked first

```
number = 1
while True:
    if number < 10:
        print(‘Still in single digits’)
        number += 1
    else:
        print(‘Double digits entered’)
        break
print(number)
```

Indentation

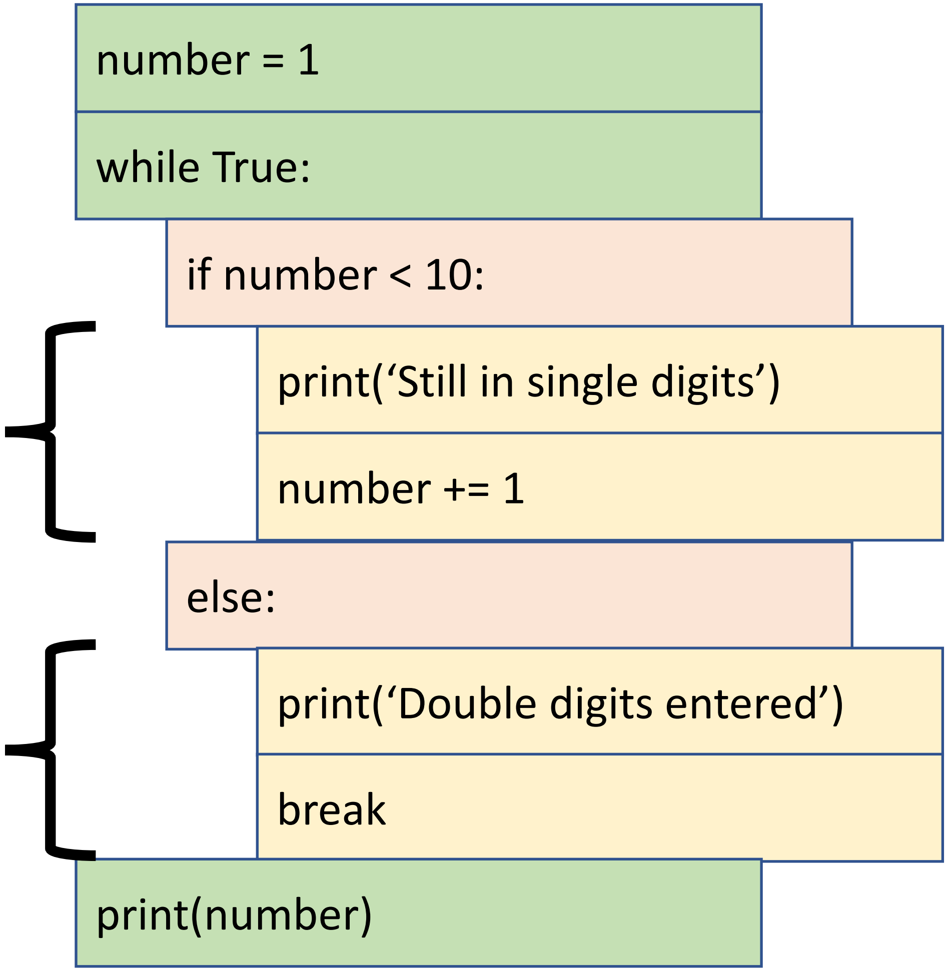


checked first

checked only if condition is false

```
number = 1
while True:
    if number < 10:
        print(‘Still in single digits’)
        number += 1
    else:
        print(‘Double digits entered’)
        break
print(number)
```

Indentation



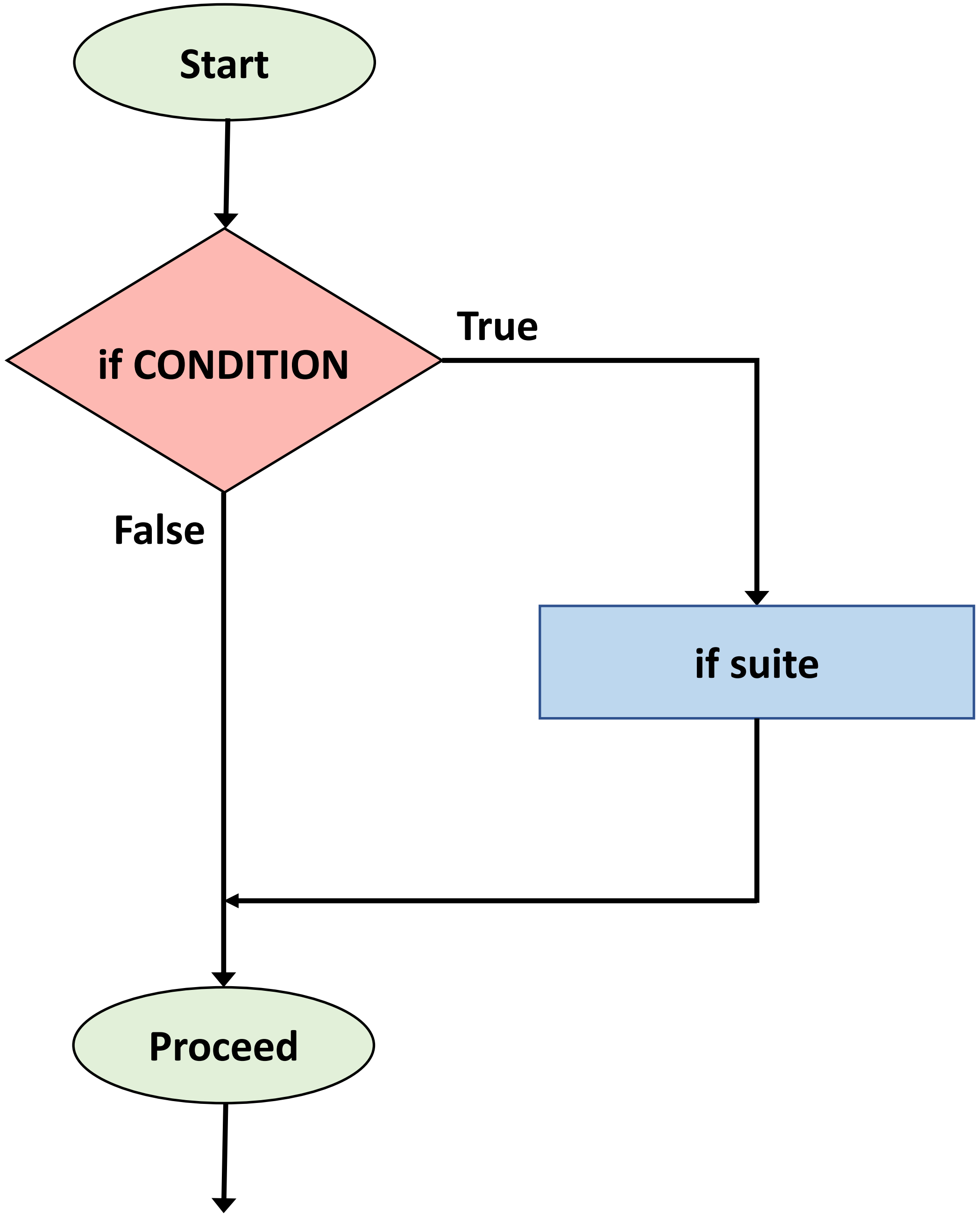
```
number = 1
while True:
    if number < 10:
        print(‘Still in single digits’)
        number += 1
    else:
        print(‘Double digits entered’)
        break
print(number)
```

Lowest priority: will only be run if the expression at next-higher priority is true

Lowest priority: will only be run if the expression is false and the else clause is used, otherwise will be skipped entirely

`if` statements

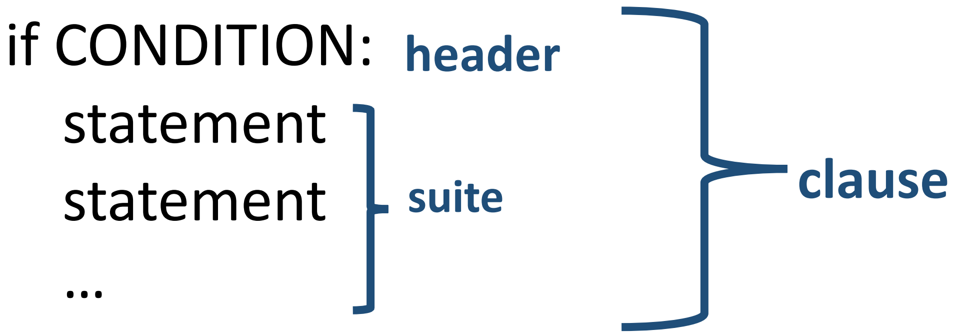
- `if` keyword creates statement for conditional execution
- If the condition is true, the code block beneath it ("suite") is executed
- If the condition is false, the suite is not executed



`if` statements

- `if` keyword creates statement for conditional execution
- If the condition is true, the code block beneath it (“suite”) is executed
- If the condition is false, the suite is not executed

if statement



```
if CONDITION:
    statement
    statement
    …
```

`if` statements

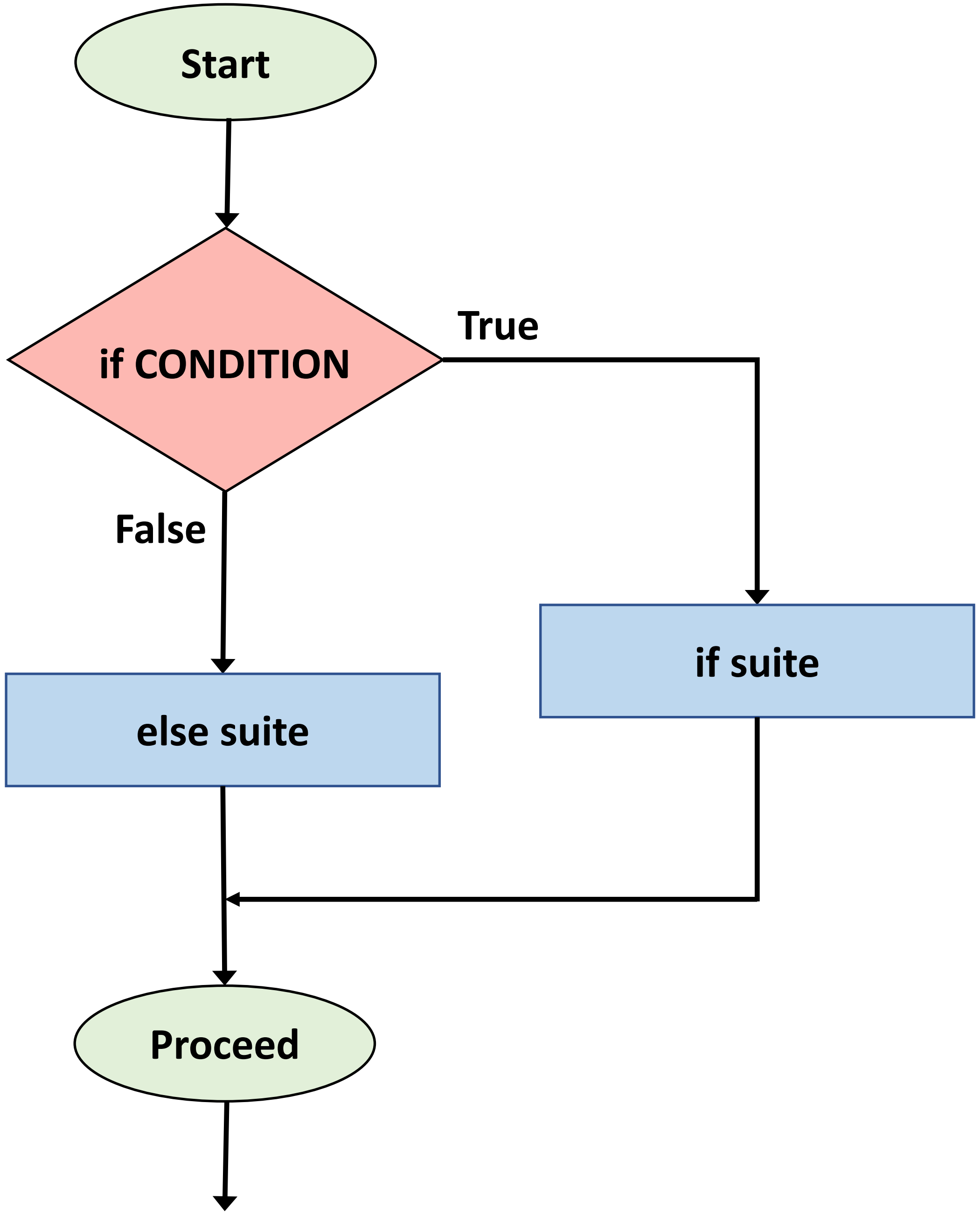
- `if` keyword creates statement for conditional execution
- If the condition is true, the code block beneath it (“suite”) is executed
- If the condition is false, the suite is not executed

```
>>> x = 10
>>> y = 5
>>> if x > y:
. . .   print(‘x greater than y’)
x greater than y

>>> if x < y:
. . .    print(‘x less than y’)
>>>
```

`if` statements

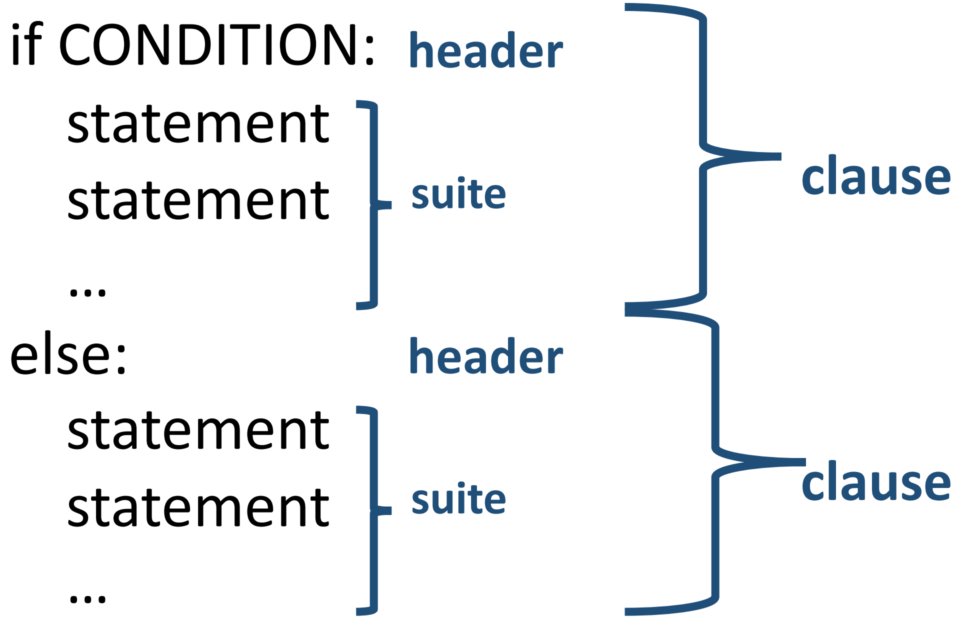
- Becomes more powerful with `else` continuation clause
- Suite below `else` clause is executed only if condition is false; otherwise, it is ignored



`if` statements

- Becomes more powerful with `else` continuation clause
- Suite below `else` clause is executed only if condition is false; otherwise, it is ignored

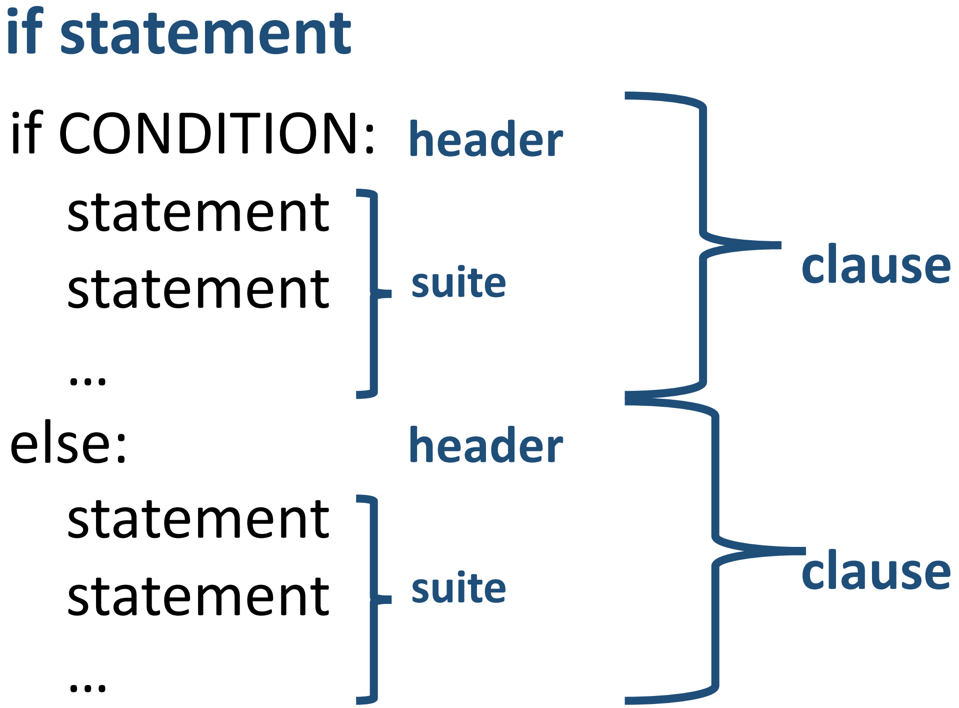
if statement



```
if CONDITION:
    statement
    statement
    …
else:
    statement
    statement
    …
```

`if` statements

- Becomes more powerful with `else` continuation clause
- Suite below `else` clause is executed only if condition is false; otherwise, it is ignored



if statement

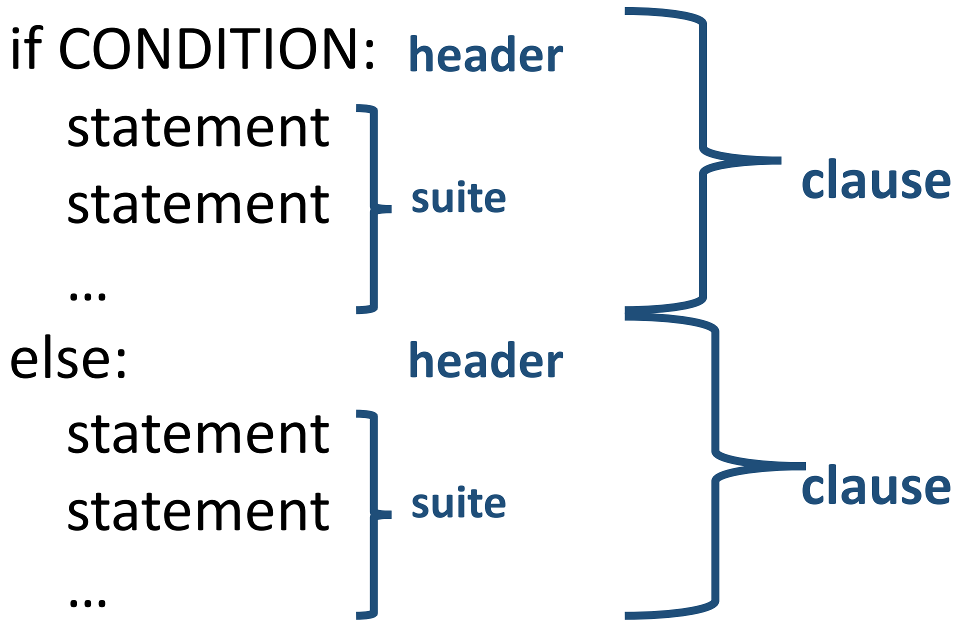
```
if CONDITION:
    statement
    statement
    …
else:
    statement
    statement
    …
```

Clause headers are at the same indentation level, so they are given the same priority

`if` statements

- Becomes more powerful with `else` continuation clause
- Suite below `else` clause is executed only if condition is false; otherwise, it is ignored

if statement



```
if CONDITION:   header
    statement
    statement   suite      clause
    …
else:           header
    statement
    statement   suite      clause
    …
```

Clause headers are at the same indentation level, so they are given the same priority

Suites are at a larger indentation level, so they are given lower priority, and one is executed only if its header is used

`if` statements

- Becomes more powerful with `else` continuation clause
- Suite below `else` clause is executed only if condition is false; otherwise, it is ignored

```
>>> x = 5
>>> y = 10
>>> if x > y:
. . .    print(‘x greater than y’)
. . . else:
. . .    print(‘x less than or equal to y’)
x less than or equal to y
```

`if` statements

- Becomes more powerful with `else` continuation clause
- Suite below `else` clause is executed only if condition is false; otherwise, it is ignored

Example 1

```
>>> x = 10
>>> y = 5
>>> if x > y:
. . .   print(‘x greater than y’)
. . . else:
. . .   print(‘x less than or equal to y’)
x greater than y
```

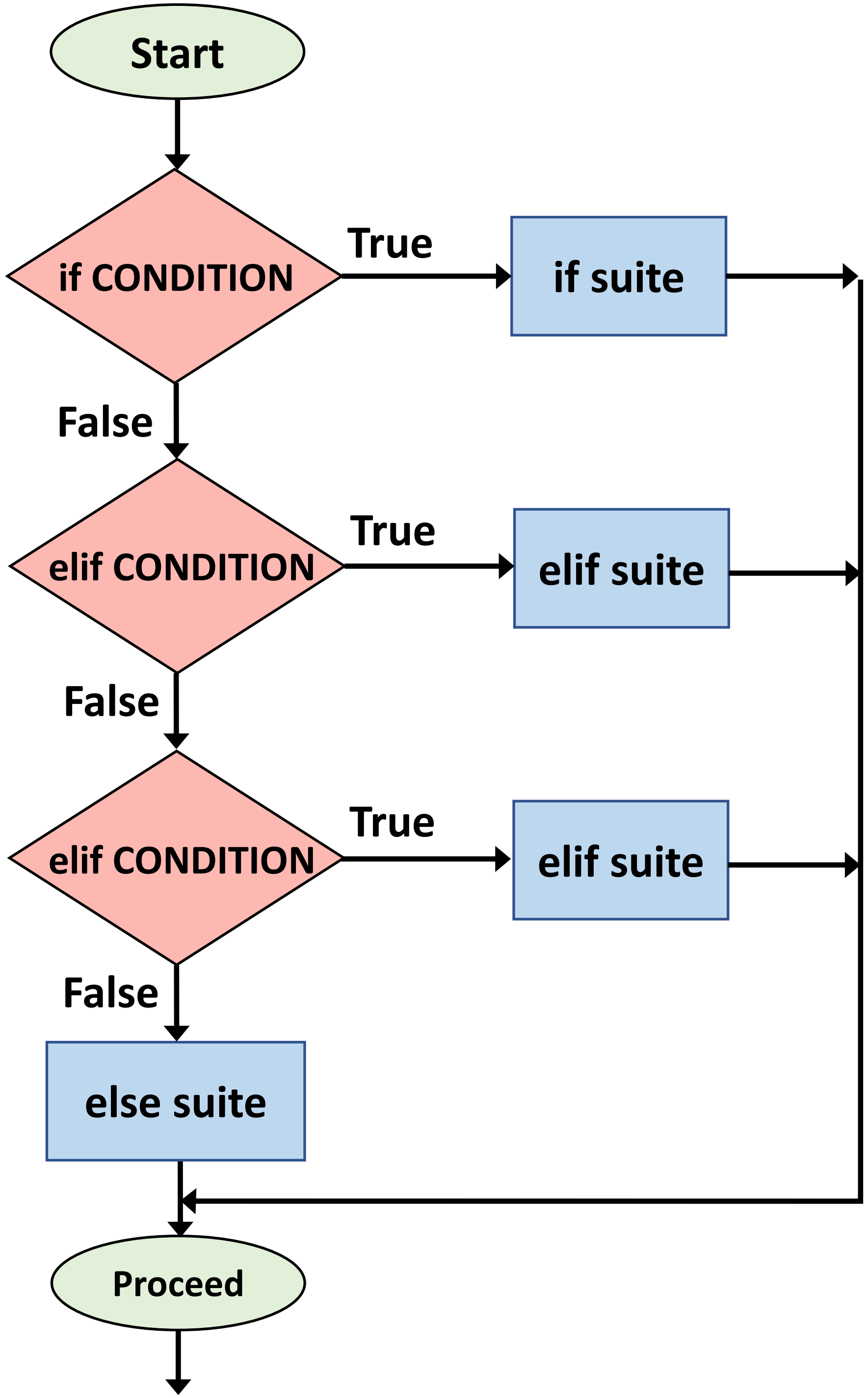
Example 2

```
>>> x = 5
>>> y = 10
>>> if x > y:
. . .   print(‘x greater than y’)
. . . else:
. . .   print(‘x less than or equal to y’)
x less than or equal to y
```

`if` statements

- Becomes even more powerful with `elif` continuation clause(s)
- 0 or more `elif` clauses can be used
- No limit to the number of `elif` clauses!

- Conditions are evaluated one-by-one in order until a true condition is found; then the corresponding suite is executed, and **no other part of the `if` statement is evaluated or executed**
- If all conditions are false, the `else` suite is executed

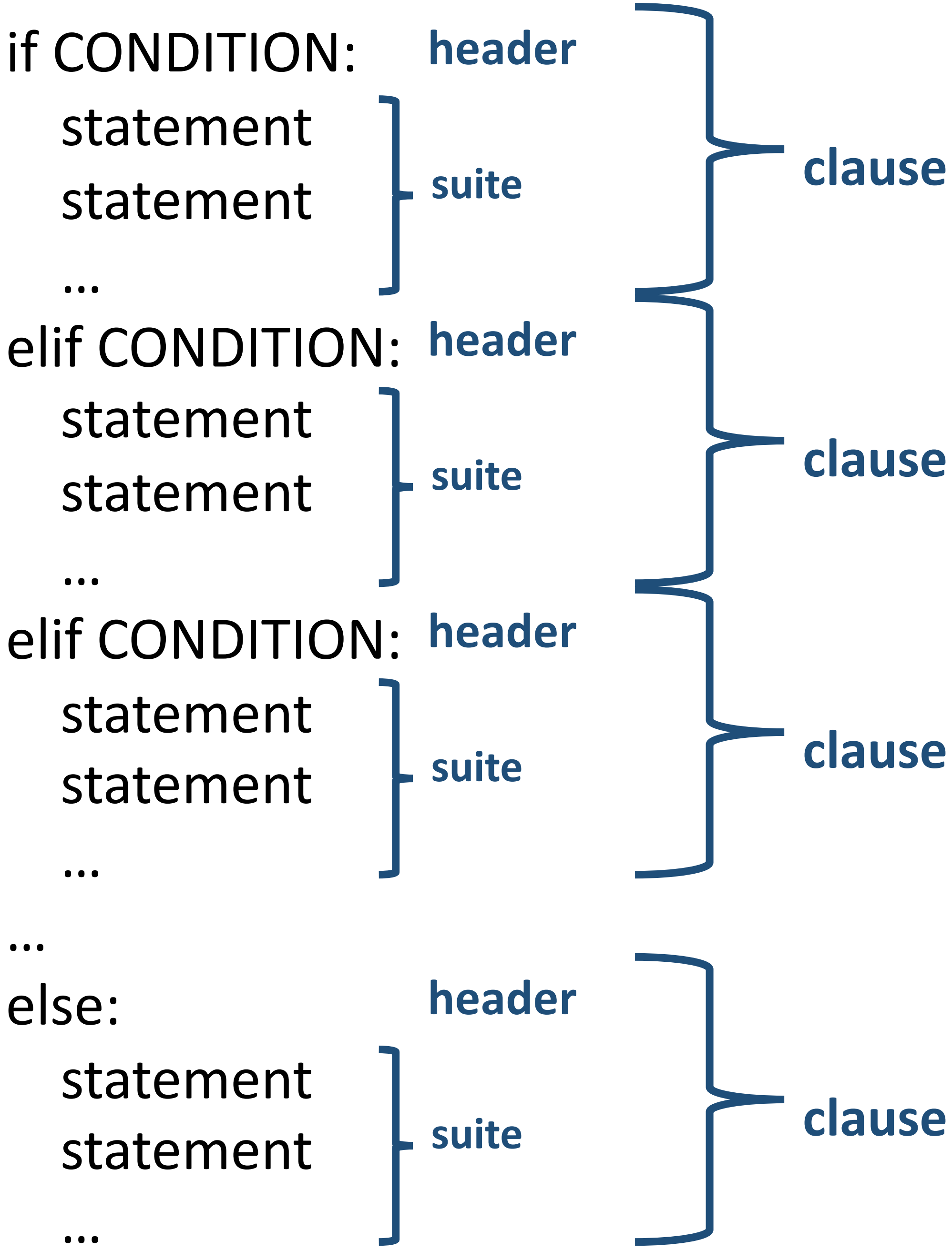


`if` statements

- Becomes even more powerful with `elif` continuation clause(s)
- 0 or more `elif` clauses can be used
- No limit to the number of `elif` clauses!

- Conditions are evaluated one-by-one in order until a true condition is found; then the corresponding suite is executed, and **no other part of the `if` statement is evaluated or executed**
- If all conditions are false, the `else` suite is executed

if statement



```
if CONDITION:
    statement
    statement
    …
elif CONDITION:
    statement
    statement
    …
elif CONDITION:
    statement
    statement
    …
…
else:
    statement
    statement
    …
```

`if` statements

- Becomes even more powerful with `elif` continuation clause(s)
- 0 or more `elif` clauses can be used
- No limit to the number of `elif` clauses!

- Conditions are evaluated one-by-one in order until a true condition is found; then the corresponding suite is executed, and **no other part of the `if` statement is evaluated or executed**
- If all conditions are false, the `else` suite is executed

```
>>> x = 5
>>> y = 10
>>> if x > y:
. . .    print(‘x greater than y’)
. . . elif x == y:
. . .    print(‘x equals y’)
. . . else:
. . .    print(‘x less than y’)
x less than y
```

`if` statements

- Becomes even more powerful with `elif` continuation clause(s)

Example 1

```
>>> x = 3
>>> y = 3
>>> if x > y:
. . .   print(‘x greater than y’)
. . . elif x == y:
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x equals y
```

Example 2

```
>>> x = 5
>>> y = 10
>>> if x > y:
. . .   print(‘x greater than y’)
. . . elif x == y:
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x less than y
```

`if` statements

- Conditions are evaluated one-by-one
- Only the suite beneath the first true condition is executed
- **All further clauses and suites ignored after a true condition is found**

Example 1

```
>>> x = 3
>>> y = 3
>>> if x > y:
. . .   print(‘x greater than y’)
. . . elif x == y:
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x equals y
```

checked first, false, continues to next condition

`if` statements

- Conditions are evaluated one-by-one
- Only the suite beneath the first true condition is executed
- **All further clauses and suites ignored after a true condition is found**

Example 1

```
>>> x = 3
>>> y = 3
>>> if x > y:
. . .   print('x greater than y')
. . . elif x == y:
. . .   print('x equals y')
. . . else:
. . .   print('x less than y')
x equals y
```

← checked first, false, continues to next condition

← checked second, true, suite executed, rest of statement ignored

`if` statements

- Conditions are evaluated one-by-one
- Only the suite beneath the first true condition is executed
- **All further clauses and suites ignored after a true condition is found**

Example 2

```
>>> x = 5
>>> y = 10
>>> if x > y:
. . .   print(‘x greater than y’)
. . . elif x == y:
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x less than y
```

checked first, false, continues to next condition

`if` statements

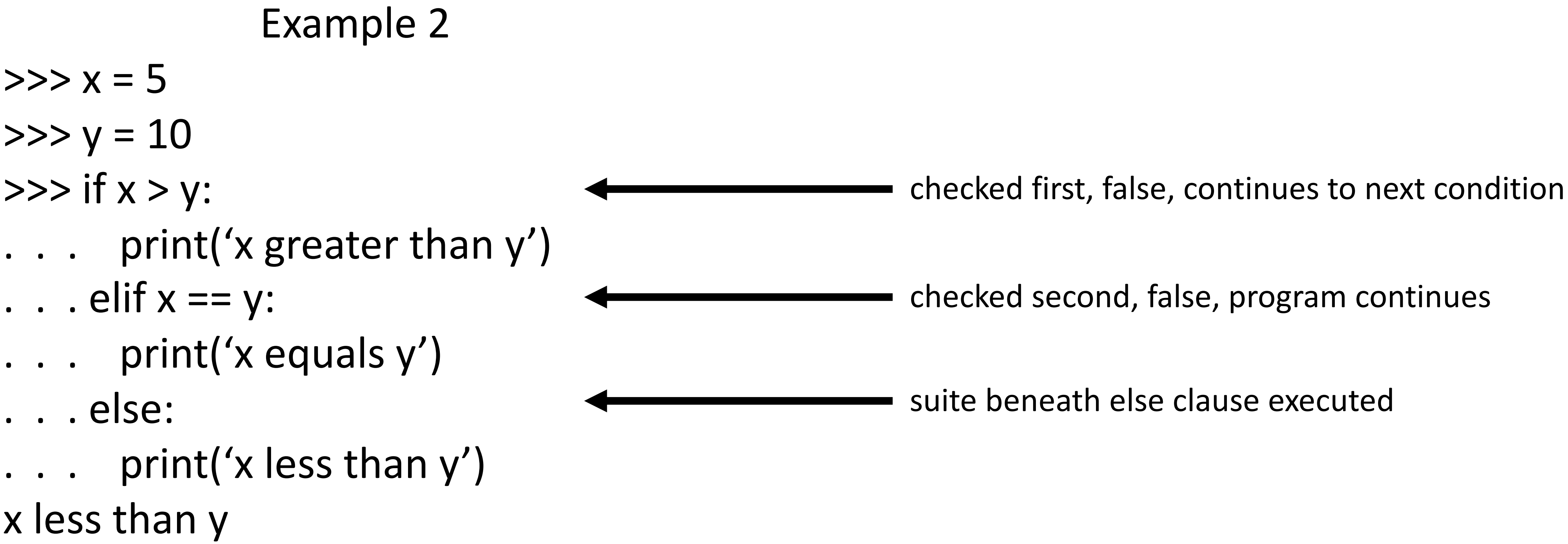
- Conditions are evaluated one-by-one
- Only the suite beneath the first true condition is executed
- **All further clauses and suites ignored after a true condition is found**

Example 2

```
>>> x = 5
>>> y = 10
>>> if x > y:                          ← checked first, false, continues to next condition
. . .   print(‘x greater than y’)
. . . elif x == y:                     ← checked second, false, program continues
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x less than y
```

`if` statements

- If all `if` and `elif` conditions are false, the suite beneath the `else` clause is executed



Example 2

```
>>> x = 5
>>> y = 10
>>> if x > y:
. . .   print(‘x greater than y’)
. . . elif x == y:
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x less than y
```

checked first, false, continues to next condition

checked second, false, program continues

suite beneath else clause executed

`if` statements

- Conditions are evaluated one-by-one
- Only the suite beneath the first true condition is executed
- **All further clauses and suites ignored after a true condition is found**

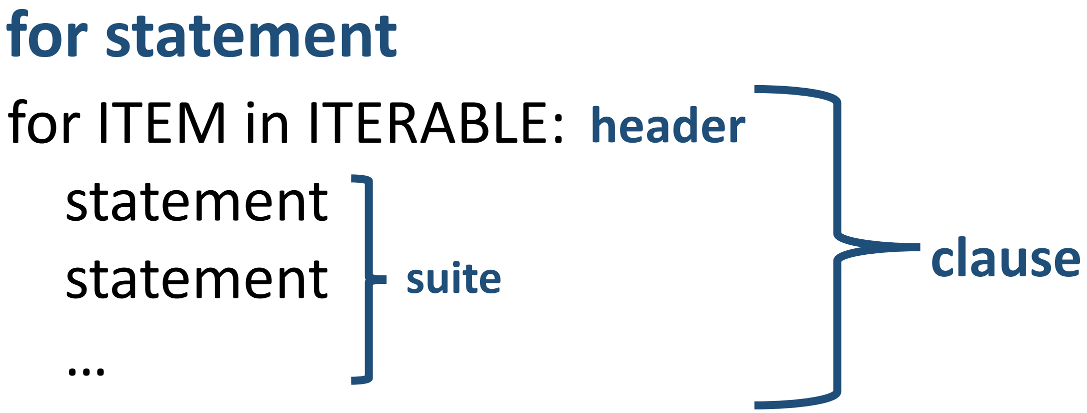
Example 3

```
>>> x = 8
>>> y = 3
>>> if x > y:
. . .   print(‘x greater than y’)
. . . elif x == y:
. . .   print(‘x equals y’)
. . . else:
. . .   print(‘x less than y’)
x greater than y
```

checked first, true, suite executed, statement stops

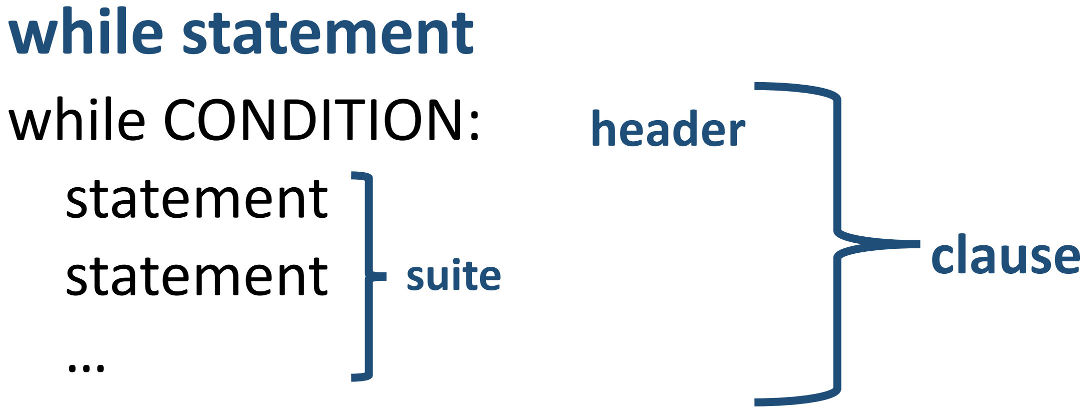
Loops

- Code is executed repeatedly
- `for` loop: suite of statement(s) executed for each item in an *iterable* object (which includes sequences)
- `while` loop: suite of statement(s) executed as long as a condition is true



for statement

for ITEM in ITERABLE: header
statement
statement
…
suite
clause

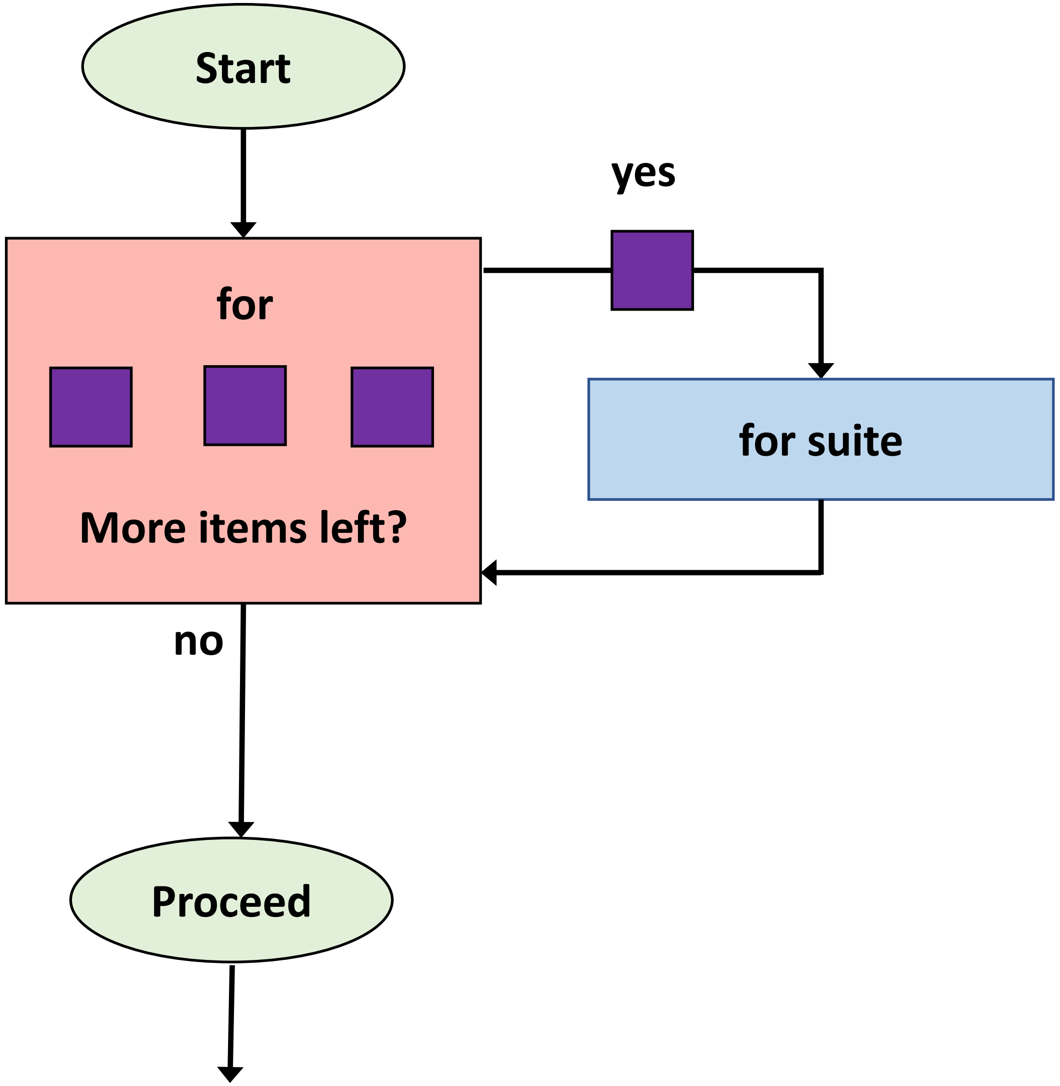


while statement

while CONDITION: header
statement
statement
…
suite
clause

`for` loops

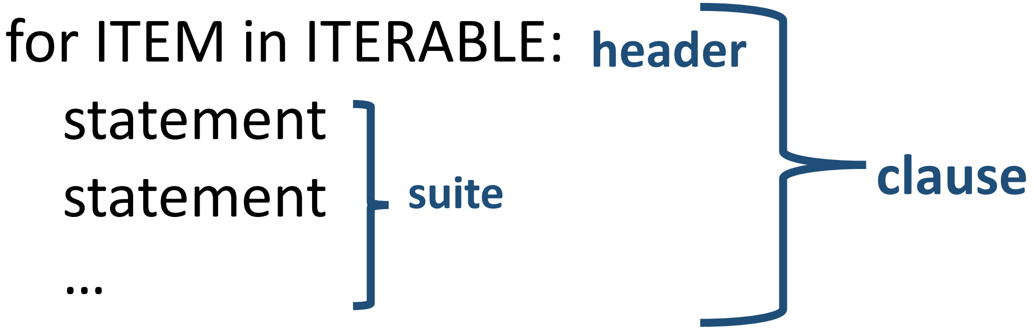
- `for` keyword creates statement for repeated execution for each item of an iterable object
- Suite beneath the `for` statement is executed for each item of an iterable object
- Loop terminates once all items are exhausted



`for` loops

- `for` keyword creates statement for repeated execution for each item of an iterable object
- Suite beneath the `for` statement is executed for each item of an iterable object
- Loop terminates once all items are exhausted

for statement



```
for ITEM in ITERABLE:
    statement
    statement
    …
```

header

suite

clause

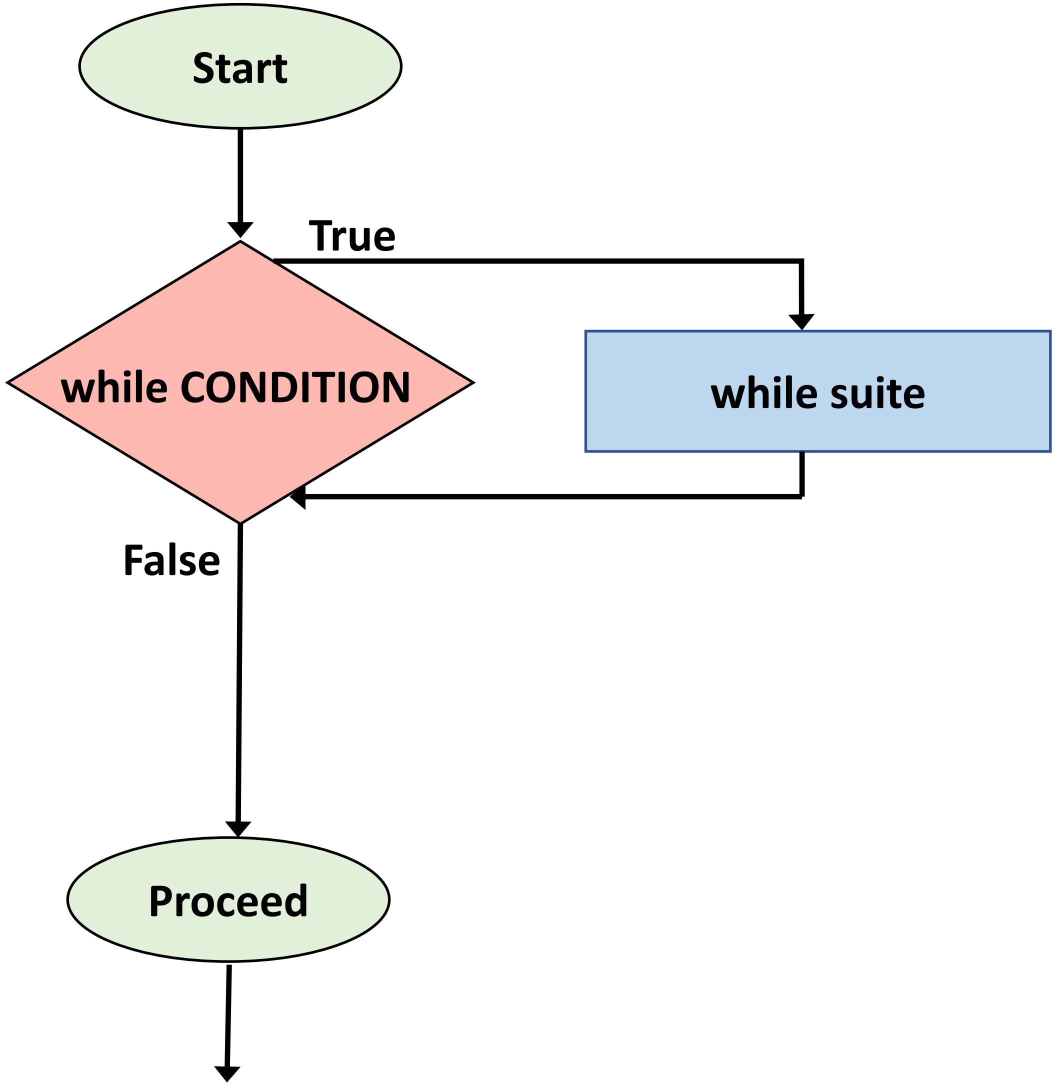
`for` loops

- `for` keyword creates statement for repeated execution for each item of an iterable object
- Suite beneath the `for` statement is executed for each item of an iterable object
- Loop terminates once all items are exhausted

```
>>> my_list = [1, 2, 3, 4]
>>> for number in my_list:
. . .    print(‘Square of ’ + str(number) + ‘is:’)
. . .    print(number**2)
Square of 1 is:
1
Square of 2 is:
4
Square of 3 is:
9
Square of 4 is:
16
>>>
```

`while` loops

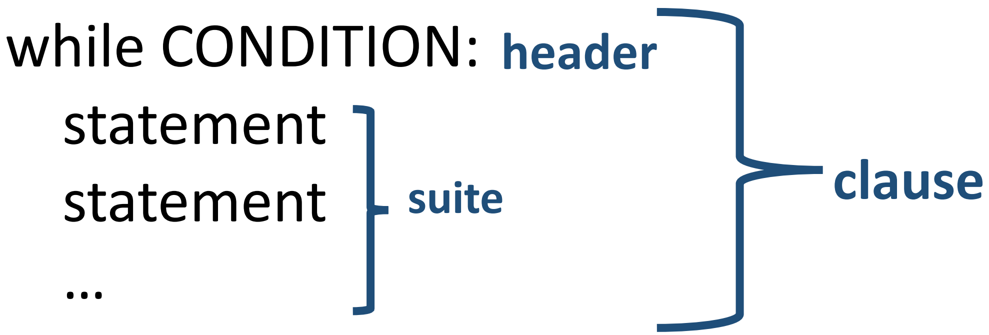
- `while` keyword creates statement for repeated execution as long as a condition is true
- Suite beneath the `while` statement is executed as long as the condition remains true
- Loop terminates when the condition becomes false



`while` loops

- `while` keyword creates statement for repeated execution as long as a condition is true
- Suite beneath the `while` statement is executed as long as the condition remains true
- Loop terminates when the condition becomes false

while statement



```
while CONDITION:
    statement
    statement
    …
```

`while` loops

- `while` keyword creates statement for repeated execution as long as a condition is true
- Suite beneath the `while` statement is executed as long as the condition remains true
- Loop terminates when the condition becomes false

```
>>> counter = 1
>>> while counter < 5:
. . .    print(counter)
. . .    counter += 1
1
2
3
4
>>>
```

`while` loops

- `while` keyword creates statement for repeated execution as long as a condition is true
- Suite beneath the `while` statement is executed as long as the condition remains true
- Loop terminates when the condition becomes false

```
>>> counter = 1
>>> while counter < 5:
. . .    print(counter)
. . .    counter += 1
1
2
3
4
>>>
```

augmented assignment

changes a number and re-assigns it to the original variable

counter = counter + 1
or
counter += 1

`break` statement

- Used to terminate a (`for` or `while`) loop

```
>>> multiples_of_three = [3, 6, 9, 12, 15]
>>> for multiple in multiples_of_three:
. . .    print(multiple)
. . .    if multiple > 10:
. . .        print(‘Breaking out of loop.’)
. . .        break
. . .    print(‘Moving onto next multiple.’)
3
Moving onto next multiple.
6
Moving onto next multiple.
9
Moving onto next multiple.
12
Breaking out of loop.
>>>
```

`break` statement

- Used to terminate a (`for` or `while`) loop

```
>>> counter = 1
>>> while counter < 5:
. . .     if counter >= 3:
. . .         print(‘Breaking out of loop.’)
. . .         break
. . .     print(counter)
. . .     counter += 1
1
2
Breaking out of loop.
>>>
```

`continue` statement

- Used to skip the rest of the suite inside a (`for` or `while`) loop for the current iteration only
- Loop continues with next iteration

```
>>> my_list = [1, 2, 3, 4, 5, 6]
>>> for num in my_list:
. . .    if num % 2 == 0: # check if even
. . .        continue
. . .    print(num)
1
3
5
>>>
```

`continue` statement

- Used to skip the rest of the suite inside a (`for` or `while`) loop for the current iteration only
- Loop continues with next iteration

```
>>> my_list = [1, 2, 3, 4, 5, 6]
>>> for num in my_list:
. . .    if num % 2 == 0: # check if even
. . .        continue
. . .    print(num)
1
3
5
>>>
```

modulo operator %

returns the remainder of the division of the first term by the second term

great for checking if a number is even or odd!

`continue` statement

- Used to skip the rest of the suite inside a (`for` or `while`) loop for the current iteration only
- Loop continues with next iteration

```
>>> counter = 1
>>> while counter < 5:
. . .    if counter <= 3:
. . .        print(counter)
. . .        counter += 1
. . .        continue
. . .    print(‘Done counting.’)
. . .    break
1
2
3
Done counting.
>>>
```

Functions

- Used to accomplish specific tasks

- *Call* to execute
    - Form of function call: **function_name(arg1, arg2, …)**
        - Any given function may require 0 or more values in place of *arguments* arg1, arg2, …
        - Read a function’s documentation to determine which (if any) arguments must be supplied and which (if any) are optional
- Exit by *returning* (passing back) value(s) to the user
    - Value(s) can be assigned to variable(s) for later use

Built-in functions

- Useful functions that always are available and are commonly used by programmers in all fields and applications

Function	Description
abs(x)	returns absolute value of number x
len(o)	returns number of items in iterable object o
list(o)	returns list constructed from items in iterable object o
max(o)	returns largest item in iterable object o
min(o)	returns smallest item in iterable object o
print(o)	displays representation of object o to screen (or other standard output device)
round(x, n)	returns x rounded to n digits precision after the decimal point
str(o)	returns string representation of object o
sum(o)	returns sum of items in iterable object o

Methods

- Functions associated with (“belonging to”) a particular type of object
- Special call format: **object.method_name(arg1, arg2, …)**

- Examples of list methods
  - list.append(x) # append item x to end of list
  - list.insert(i, x) # insert item x at index i
  - list.pop(i) # remove item at index i
  - list.count(x) # return the number of times item x appears in the list
  - list.sort() # sort the items of the list in-place
  - list.reverse() # reverse the items of the list in-place

User-defined functions

- We can *define* our own functions and then call them to carry out specific tasks!
- Users define their own functions to follow the DRY principle
  - “Don’t Repeat Yourself”
  - Don’t repeat code: Write reusable code instead!

User-defined functions

- Function definition
  - Header: `def` keyword followed by function name and 0 or more parameters in parentheses followed by colon
  - Function body (suite) indented
- Body usually includes a `return` statement
  - Returns None if no explicit `return` statement included, or if no value(s) provided after the `return` keyword
- Execution of the function stops when a `return` statement is run, and no code after it within the body is executed

```
def function_name(param1, param2, …):
    statement
    statement
    …
    return value(s)
```

User-defined functions

- Function definition
  - Header: `def` keyword followed by function name and 0 or more parameters in parentheses followed by colon
  - Function body (suite) indented
- Body usually includes a `return` statement
  - Returns None if no explicit `return` statement included, or if no value(s) provided after the `return` keyword
- Execution of the function stops when a `return` statement is run, and no code after it within the body is executed

```
>>> def simple_func(x):
. . .    print(‘Squaring in progress.’)
. . .    return x**2
>>> simple_func(6)
Squaring in progress.
36
>>> simple_func(-4)
Squaring in progress.
16
```

User-defined functions

- Function definition
  - Header: `def` keyword followed by function name and 0 or more parameters in parentheses followed by colon
  - Function body (suite) indented
- Body usually includes a `return` statement
  - Returns None if no explicit `return` statement included, or if no value(s) provided after the `return` keyword
- Execution of the function stops when a `return` statement is run, and no code after it within the body is executed

```
>>> def testing(x):
. . .     print(‘Printing for the first time.’)
. . .     return x
. . .     x += 1
. . .     print(‘Printing for the second time.’)
. . .     return x
>>> testing(2)
Printing for the first time.
2
>>> testing(7)
Printing for the first time.
7
```

User-defined functions

- Returned value(s) can be assigned to variables for later use

```
>>> def simple_func(x):
. . .    return x**2
>>> my_square = simple_func(6)
>>> my_square
36
>>> my_square = simple_func(-4)
>>> my_square
16
```

User-defined functions

- *Default parameter values* allow arguments to be omitted from a call

```
>>> def simple_func(x):
. . .    return x**2
>>> simple_func(5)
25
>>> simple_func()
TypeError: simple_func() missing 1 required positional argument: 'x'
```

User-defined functions

- *Default parameter values* allow arguments to be omitted from a call

```
>>> def simple_func(x=2):
. . .    return x**2
>>> simple_func(5)
25
>>> simple_func()
4
```

Scope

- *Global variables* are those created outside functions
  - They can be read by everything inside your program!
  - They can be read inside functions without being passed in as arguments

```
>>> x = 3.5
>>> def simple_func():
. . .     return x
>>> simple_func()
3.5
```

Scope

- *Global variables* are those created outside functions
  - They can be read by everything inside your program!
  - They can be read inside functions without being passed in as arguments
  - But they cannot be modified inside functions…

```
>>> x = 3.5
>>> def simple_func():
. . .     x += 2 # add 2 to x and reassign it
. . .     return x
>>> simple_func()
UnboundLocalError: local variable ‘x’
referenced before assignment
```

Scope

- *Global variables* are those created outside functions
  - They can be read by everything inside your program!
  - They can be read inside functions without being passed in as arguments
  - But they cannot be modified inside functions…unless they are passed in as arguments and the modified value is returned

```
>>> x = 3.5
>>> def simple_func(x):
. . .    x += 2 # add 2 to x and reassign it
. . .    return x
>>> simple_func(x)
5.5
```

Scope

- *Global variables* are those created outside functions
    - They can be read by everything inside your program!
    - They can be read inside functions without being passed in as arguments
    - But they cannot be modified inside functions…unless they are passed in as arguments and the modified value is returned
    - For the new value to “stick,” the returned value must be assigned to the variable

```
>>> x = 3.5
>>> def simple_func(x):
. . .     x += 2 # add 2 to x and reassign it
. . .     return x
>>> simple_func(x)
5.5
>>> x
3.5
```

Scope

- *Global variables* are those created outside functions
  - They can be read by everything inside your program!
  - They can be read inside functions without being passed in as arguments
  - But they cannot be modified inside functions…unless they are passed in as arguments and the modified value is returned
  - For the new value to “stick,” the returned value must be assigned to the variable

```
>>> x = 3.5
>>> def simple_func(x):
. . .     x += 2 # add 2 to x and reassign it
. . .     return x
>>> simple_func(x)
5.5
>>> x
3.5
>>> x = simple_func(x)
>>> x
5.5
```

Scope

- *Local variables* are those created inside functions when the functions are called
  - They can be read and modified only inside the function within which they are created

```
>>> x = ‘global’
>>> def simple_func():
. . .     x = ‘local’
. . .     y = ‘another local’
. . .     print(x)
>>> simple_func()
local
>>> print(x)
global
>>> y
NameError: name ‘y’ is not defined
```

Scope

- *Local variables* are those created inside functions when the functions are called
  - They can be read and modified only inside the function within which they are created
  - Their value must be returned in order to access it outside the function

```
>>> x = ‘global’
>>> def simple_func():
. . .     x = ‘local’
. . .     return x
>>> simple_func()
‘local’
>>> x
‘global’
```

Scope

- *Local variables* are those created inside functions when the functions are called
    - They can be read and modified only inside the function within which they are created
    - Their value must be returned in order to access it outside the function
    - Their returned value can be assigned to a variable, making it global in scope

```
>>> x = ‘global’
>>> def simple_func():
. . .     x = ‘local’
. . .     return x
>>> simple_func()
‘local’
>>> x
‘global’
>>> x = simple_func()
>>> x
‘local’
```

Structured programming

- Use control flow to produce optimal code
    - Readable
    - Reusable
    - Efficient
    - Easier and quicker to develop
    - Easier and quicker to modify

- Guided by principles of
    - Sequence—order of execution
    - Selection—conditional execution
    - Iteration—repeated execution

Structured programming

- Use control flow to produce optimal code
  - Readable
  - Reusable
  - Efficient
  - Easier and quicker to develop
  - Easier and quicker to modify

- Guided by principles of

• Sequence—order of execution	**order of statements and function calls**
• Selection—conditional execution	**`if` statements**
• Iteration—repeated execution	**`for` and `while` loops**

Guess-A-Word game

- A player solves a 5-letter word by guessing one letter at a time
- Correct answers are added to the word
- The player is allowed up to 5 wrong guesses
- If the player guesses the word without making 5 wrong guesses, they win!

_____ H _____ _____ E

Guess-A-Word game

- What are the steps in the game?
- What are actions that must be taken for the game to proceed?