

COMP47670

Introduction to Python Part 1

Slides by Derek Greene

UCD School of Computer Science
Spring 2019

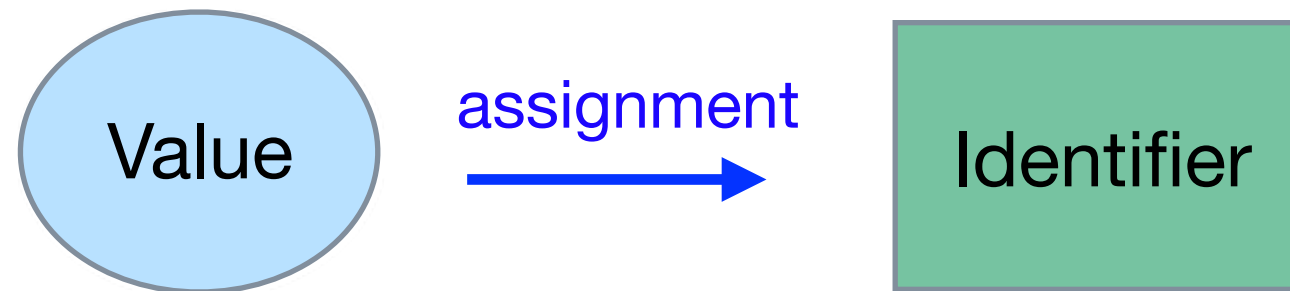


Overview

- Python Basics
 - Variables and Types
 - Operators
 - Expressions and Statements
 - Calling Functions
- Data Structures
 - Lists, Tuples, Dictionaries, Sets
- Blocks and Indentation
- Flow Control
 - Conditionals, Loops

Variables in Python

- **Variable:** A container in memory, which has a unique name, where you can store a value.



- In Python we do not need to define variables in advance. Create a variable `x` by assigning it a value, where the `=` symbol denotes **assignment**.

```
x = 100
```

Variable
naming
rules:

- Can contain both letters and numbers, but must begin with a letter.
- Can contain the underscore character.
- Should not clash with reserved keywords.

Note: Python is a case sensitive language!

Variable Types

- Each variable also has a **type**, indicating the nature of the value that it stores.
- Basic Python variable types:

Integer

```
a = 3  
b = -125
```

Floats

```
score = 0.452  
result = 1e-5
```

Can use decimal or scientific notation

Boolean

```
answer = True  
test_val = False
```

True or False. Can also use '1' and '0'

String

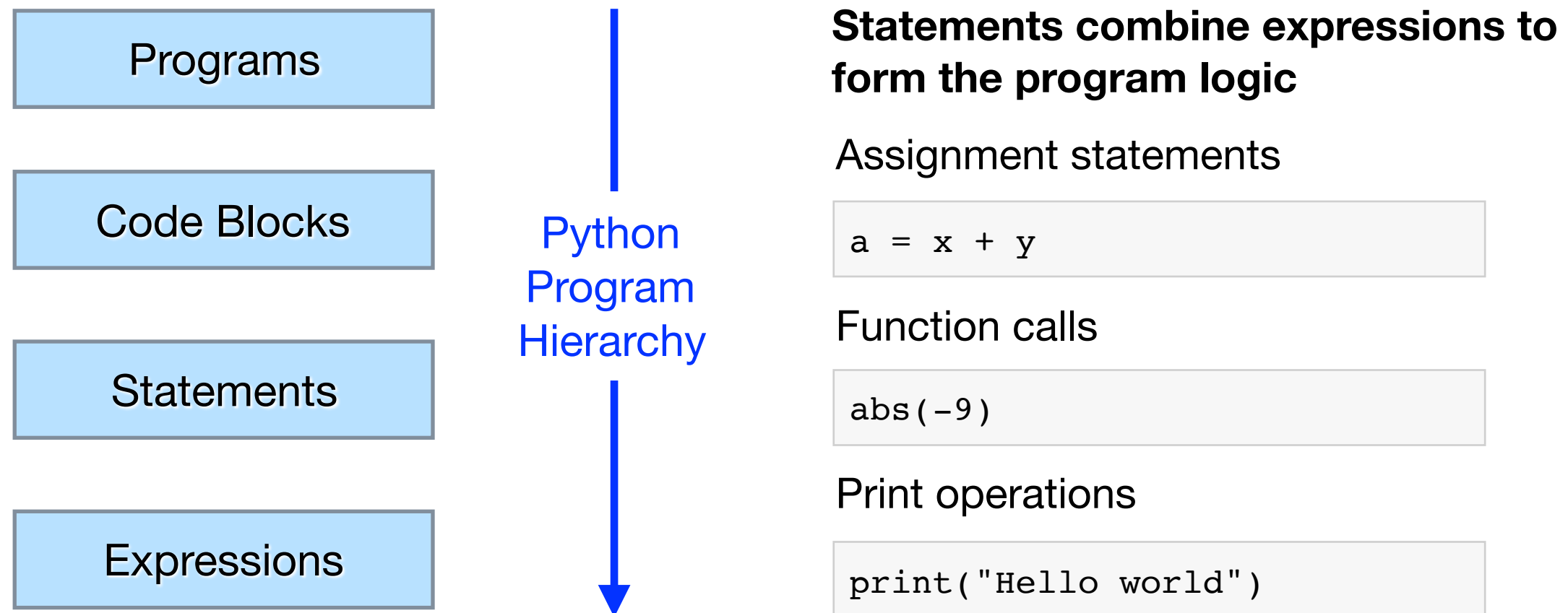
```
mytext = "this is some text"  
s2 = 'more text'  
hello_str = "hello" + " world"
```

Can use either single or double quotes - but be consistent!

- Also **None** is the special empty or "null" value

Expressions and Statements

- Combinations of operators and variables/values to create **expressions**, which are the building blocks of Python code.



- Python contains a variety of different unary and binary operators for different purposes.

Arithmetic Operators

- Python can be used as a simple calculator. It supports all basic mathematical operators, such as $+$, $-$, $*$, $/$

```
10 * 2 - 5
```

```
15
```

```
x = 3 * 5  
print(x)
```

```
15
```

We can use the *print()* function to display the value of a variable

- Parentheses can be used to control the order in which several operators are applied.

```
4 + 10 / 2
```

```
9.0
```

```
(4 + 10) / 2
```

```
7.0
```

Normally division takes precedence over addition

- We can also use operators to perform assignment and an operation on the same variable.

```
x += 2
```

Add 2 to current value of *x*, reassign to *x*

```
x *= 10
```

Multiply current value of *x* by 10, reassign to *x*

Comparison Operators

- Any value or variable can be tested for a **truth value**. These will output a value of `True` or `False`, depending on the values (operands) are testing. A range of different comparison operators can be used to perform these tests:

Operator	Description
<code>==</code>	If the values of two operands are equal, then the output is <code>True</code> .
<code>!=</code>	If values of two operands are not equal, then the output is <code>True</code> .
<code><></code>	If values of two operands are not equal, then the output is <code>True</code> .
<code>></code>	If the value of left operand is greater than the value of right operand, then the output is <code>True</code> .
<code><</code>	If the value of left operand is less than the value of right operand, then the output is <code>True</code> .
<code>>=</code>	If the value of left operand is greater than or equal to the value of right operand, then the output is <code>True</code> .
<code><=</code>	If the value of left operand is less than or equal to the value of right operand, then the output is <code>True</code> .

Comparison Operators

- Any value or variable can be tested for a **truth value**. These will yield a value of `True` or `False`, depending on what we are testing.
- A range of different comparison operators can be used to perform these tests - e.g. equality, inequality, greater than etc.

```
y = 80
```

```
y == 80  
True
```

equals

```
y > 10  
True
```

greater
than

```
y >= 80  
True
```

greater than
or equals

```
y != 80  
False
```

not
equals

```
y < 50  
False
```

less
than

```
y <= 80  
True
```

less than
or equals

- Other operators are provided, such as modulus (%) which is used for finding the remainder on division:

```
10 % 2  
0
```

```
10 % 3  
1
```

```
10 % 4  
2
```


Boolean Operators

- Python contains operators to create more complex logical tests:

<code>not x</code>	Evaluates to True if x is False. Evaluates to False if x is True
<code>x and y</code>	If both x and y are True then evaluate to True, otherwise False
<code>x or y</code>	If either x or y are True then evaluate to True, otherwise False

`not True`

False

`True and True`

True

`True or False`

True

`5 < 10 and 5 < 3`

False

`not False`

True

`True and False`

False

`False or False`

False

`5 < 10 or 5 < 3`

True

Calling Functions

- **Functions** in Python are groups of code that have a name and can be called using parentheses notation:

Function name Argument values, in parenthesis

Return value → `res = pow(4, 3)` e.g. call function `pow()` to calculate 4 raised to the power of 3

- Not all functions require arguments or return a value.

```
print("Hello world!")
```

The function `print()` has no return value

- Python contains a large number of built-in functions that can be called in this way: <https://docs.python.org/3/library/functions.html>

- Variables in Python can also have functions associated with them. These are called using dot notation:

```
s = "TEST"
t = s.lower()
print(t)
```

Call function `lower()` associated with `s`

```
test
```

Data Structures: Lists

- **Lists:** An ordered collection of items. The items can have different types.
- Items in a list are accessed by specifying an index (position) using square bracket notation. Indexes start from 0 in Python
- As well as positional indexing, lists support a more general form of indexing known as **slicing**, which can extract an entire sub-list.
- Format: `x[i:j]` means “give me everything in x from position i up to but not including position j”.

```
mylist = [12, 108, 23]
mylist2 = ["text", 7, 0.34, True]
```

```
mylist[0]
```

```
12
```

First item has index 0

```
mylist[-1]
```

```
23
```

Access in reverse via negative numbers

```
l = [7, 6, 8, 9, 1]
l[0:2]
```

```
[7, 6]
```

```
l = [7, 6, 8, 9, 1]
l[2:5]
```

```
[8, 9, 1]
```

```
l = [7, 6, 8, 9, 1]
l[1:]
```

```
[6, 8, 9, 1]
```

```
l = [7, 6, 8, 9, 1]
l[:2]
```

```
[7, 6]
```

Default for i is 0, default for j is the end of the string

Data Structures: Lists

- Lists can be modified after creation, either by changing existing items, or adding items to the end using the `append()` function.

```
mx = [12, 33, 21]
mx[1] = 108
print(mx)
```

```
[12, 108, 21]
```

Change the 2nd
item - i.e. index 1

```
mx = [12, 33, 21]
mx.append(9)
mx.append(13)
print(mx)
```

```
[12, 33, 21, 9, 13]
```

Add two new
values to the end
of the list

- We can remove the first occurrence of a value from a list using the `remove()` function:

```
x = ["a", "b", "c"]
x.remove("b")
print(x)
```

```
['a', 'c']
```

- Multiple lists can also be concatenated together to create new lists:

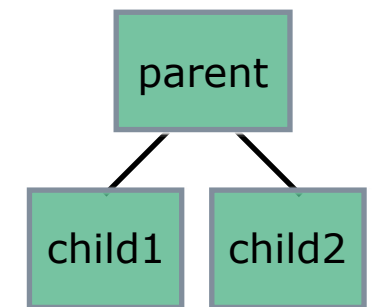
```
[8, 11] + [2, 2] + [0]
```

```
[8, 11, 2, 2, 0]
```

Data Structures: Lists

- **Nesting:** Lists can be contained inside other lists.
- Values in nested lists can be accessed using multiple indexes in square brackets:

```
child1 = [12, 108, 23]
child2 = [99, 4]
parent = [child1, child2]
```



```
parent[1][0]
```

```
99
```

```
parent[0][2]
```

```
23
```

- A variety of built-in functions can be used with lists:

```
x = ["c", "a", "b"]
```

```
len(x)
```

```
3
```

Get number of
items in a list

```
x = ["c", "a", "b"]
```

```
x.reverse()
```

```
print(x)
```

```
['b', 'a', 'c']
```

Reverse the order of
a list

```
x.sort()
```

```
print(x)
```

```
['a', 'b', 'c']
```

Sort a list in place

```
x.clear()
```

```
print(x)
```

```
[]
```

Remove all items
from a list

Data Structures: Tuples

- **Tuples** are sequences like lists but are "immutable" - i.e. they cannot be changed once they are created.
- They can provide an "integrity constraint" when passing values around a complex program.

```
mycard = ("hearts", "king")
```

Use parenthesis notation to create a tuple

```
print( mycard[1] )
```

Positional indexing works the same way as for a list

```
king
```

```
t = (2, True, "ucd")
```

Values can take different types, like a list

- Trying to modify a tuple after creation will cause an error...

```
mycard[1] = "queen"
```

```
TypeError: 'tuple' object does  
not support item assignment
```

```
t[0] = 12
```

```
TypeError: 'tuple' object does  
not support item assignment
```


Data Structures: Sets

- **Sets:** Unordered lists which contain no duplicate values.
- Can be created from lists, strings or any other iterable value, using the `set ()` function.

Convert specified list into a set.
Duplicate values are removed.

```
mylist = [1,3,1,4,3,6,8,1,4,4]
x = set(mylist)
print(x)

{1, 3, 4, 6, 8}
```

The `in` operator can be used to
check if a value is in a set.

```
1 in x
True
```

```
7 in x
False
```

- A number of functions can be used to compare two sets:

```
x = set([1,2,3,4])
y = set([3,4,5])
```

```
x.intersection(y)

{3, 4}
```

Values in both
x and y

```
x.union(y)

{1, 2, 3, 4, 5}
```

Values in either
x and y

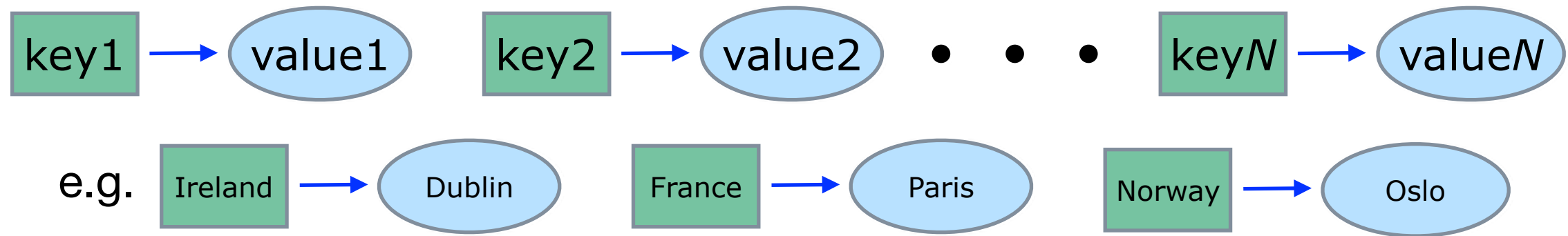
```
x.difference(y)

{1, 2}
```

Values in x
but not y

Data Structures: Dictionaries

- Data structure containing an unordered map of (**key**,**value**) pairs



Create an empty dictionary

```
d0 = {}
```

Create and populate a new dictionary

```
d1 = {"Ireland": "Dublin", "France": "Paris"}
```

Access a value in a dictionary

```
d1["Ireland"]
```

```
Dublin
```

Trying to access non-existent key causes an error

```
d1["Norway"]
```

```
KeyError: 'Norway'
```

Check for the presence of a key using the `in` operator

```
"Ireland" in d1
```

```
True
```

```
"Norway" in d1
```

```
False
```

Data Structures: Dictionaries

- Dictionaries can be modified after creation. New pairs can easily be added using the assignment operator:

```
d1["Italy"] = "Rome"
```

```
{'France': 'Paris', 'Italy': 'Rome', 'Ireland': 'Dublin'}
```

- Dictionaries have various associated functions to access the keys and/or values.

Get only the keys
from a dictionary

```
d1.keys()
```

```
dict_keys(['France', 'Italy', 'Ireland'])
```

Get only the values
from a dictionary

```
d1.values()
```

```
dict_values(['Paris', 'Rome', 'Dublin'])
```

Get all (key,value)
pairs as tuples

```
d1.items()
```

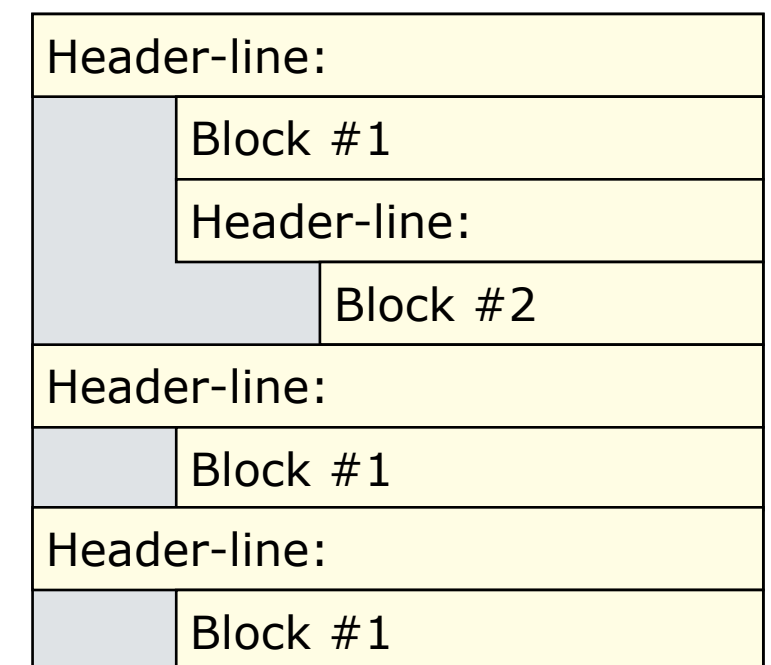
```
dict_items([('France', 'Paris'), ('Italy', 'Rome'), ('Ireland', 'Dublin')])
```

Blocks and Indentation

- **Block:** A collection of contiguous statements to be executed one after another.
- **Indentation:** Tabs or spaces appearing at the start of a line of code.
- Unlike most programming languages, indentation is significant in Python - it is used to denote the beginning and end of block structures....

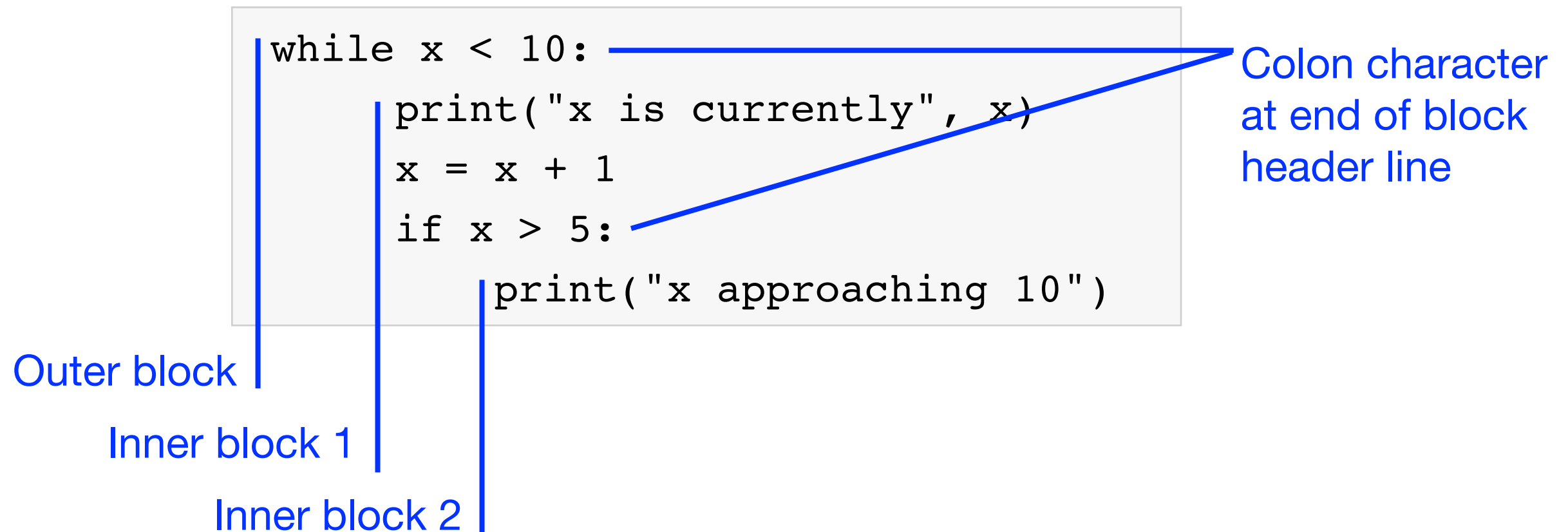
- **Task A**
 - Some actions
 - Sub-task
 - Some actions
- **Task B**
 - Some actions
- **Task C**
 - Some actions

Use indentation to denote
nested blocks of code.



Blocks and Indentation

- In Python there is no need to mark block boundaries.
 - Python automatically detects these based on line indentation.
- The indentation level of statements is significant.
 - The exact amount of indentation does not matter, only the relative indentation of the blocks.
- Makes code less verbose and improves readability.



Flow Control: Conditionals

- **If statements:** Select code to execute based on a specified Boolean expression (i.e. is the expression `True` or `False`).
- If the condition is true, we run a block of statements (called an *if-block*), else we process another block of statements (called an *else-block*). The else clause is optional.

General Format
<pre>if <condition1>: <block1> elif <condition2>: <block2> else: <block3></pre>

First branch

Alternative
branch

"Last resort"

```
x = 3  
y = 4  
if x < y:  
    print("y is greater")  
elif x == y:  
    print("values are equal")  
else:  
    print("x is greater")  
  
y is greater
```


Flow Control: Conditionals

- **If statements:** Select code to execute based on a specified Boolean expression (i.e. is the expression `True` or `False`).

```
if guess == answer:
    print("Congratulations. You guessed it.")
elif guess < answer:
    print("No. It is a higher than that.")
else:
    print("No. It is a lower than that.")
print("Done.")
```

Block for first branch

Block for second branch

Block for third branch

This block of code always gets executed

Conditions can be combined using logical operators:

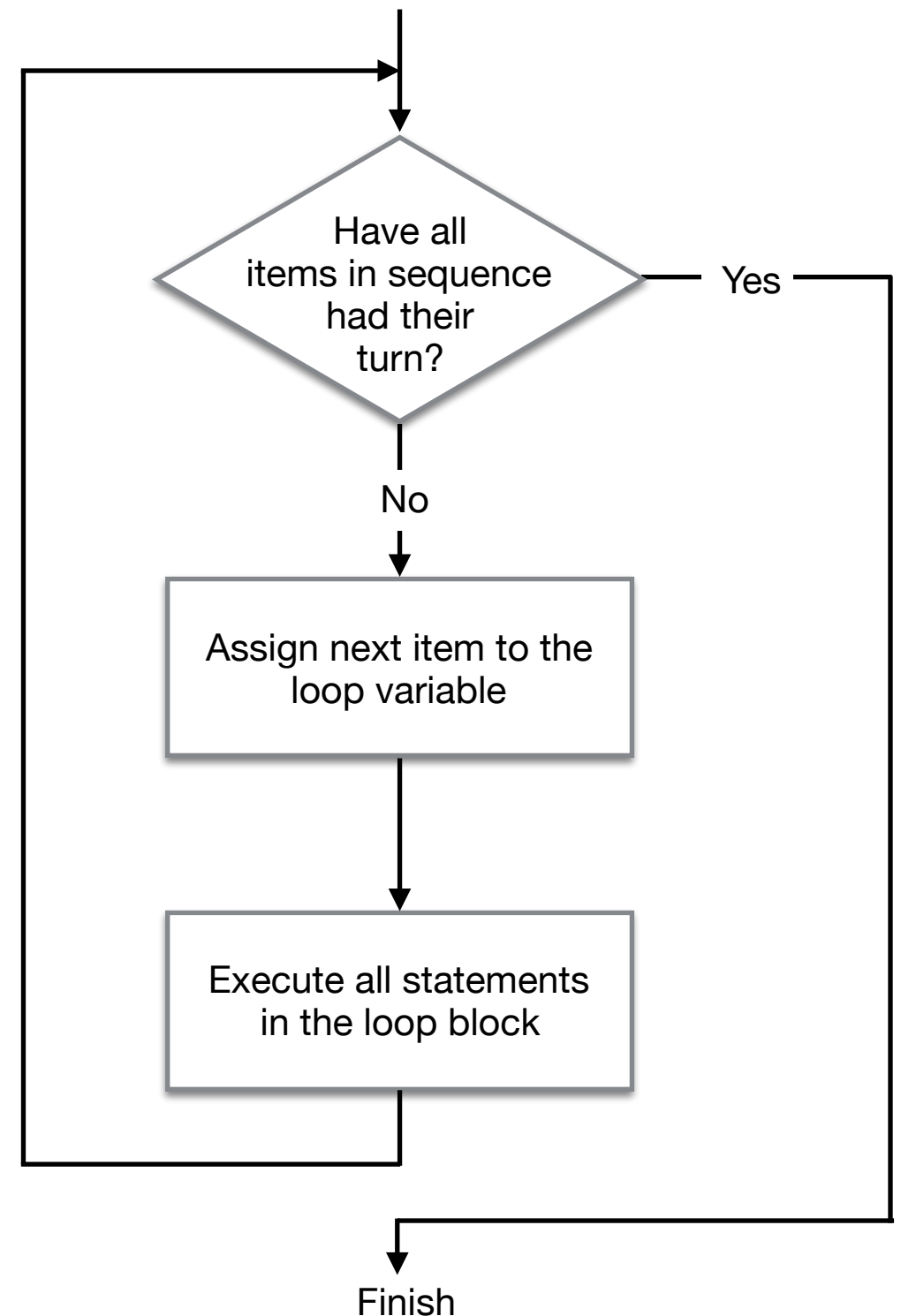
```
if x < 3 and y > 7:
    print("Both true")
```

```
if ( x < 3 and y > 7 ) or x == y:
    print("Evaluated to true")
```

Flow Control: For Loops

- **For loops:** Iterate over a sequence of items, and repeatedly apply a block of code to each item in the sequence.
- Loops can be applied to the items in any sequence or other iterable object
e.g. lists, tuples, sets, or functions that generate sequences

General Format
<pre>for <value> in <sequence>: <block></pre>



Flow Control: For Loops

- Loops can be applied to the items in any ordered sequence or other iterable object - e.g. lists, tuples, sets, or functions that generate sequences

Variable *v* takes value of each item in the list *names* →

```
names = ["John", "Mark", "Jane"]  
for v in names:  
    print("Hello", v)
```

Indented block of code to apply to each element

```
Hello John  
Hello Mark  
Hello Jane
```

Iterate over the keys in the dictionary →

```
capitals = {"Ireland": "Dublin", "France": "Paris"}  
for country in capitals:  
    print( country, "->", capitals[country] )
```

```
Ireland -> Dublin  
France -> Paris
```

Flow Control: For Loops

- Generating lists with a specific number of integers is a common task, particularly when using a For loop. The `range()` function can easily generate sequences of numbers.

Passing a single argument n gives n values starting at 0

```
for x in range(4):  
    print("Value is", x)
```

```
Value is 0  
Value is 1  
Value is 2  
Value is 3
```

Passing 2 arguments specifies the starting value and the value to stop before.

```
for x in range(11,15):  
    print("Value is", x)
```

```
Value is 11  
Value is 12  
Value is 13  
Value is 14
```

Passing three arguments specifies the starting value, value to stop before, and the step size.

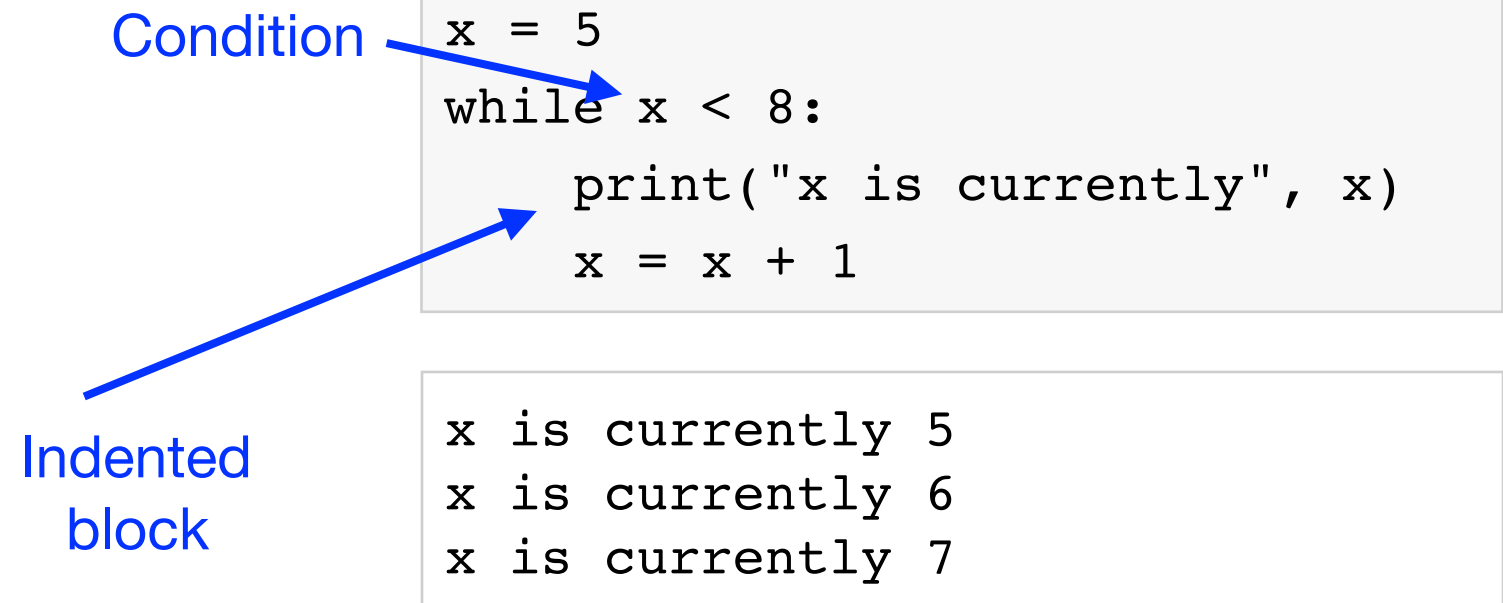
```
for x in range(5,12,3):  
    print("Value is", x)
```

```
Value is 5  
Value is 8  
Value is 11
```

Flow Control: While Loops

- **While loops:** Repeatedly execute a block of code while a specified Boolean expression still evaluates to True.

General Format
<pre>while <condition>: <block></pre>



- Special keywords can be used to control the execution of a loop...
 - **break:** Jump out of enclosing loop
 - **continue:** Jump to top of enclosing loop
 - **pass:** Do nothing (empty statement).

```
while x < 8:  
    if x%2 == 0:  
        break  
    x = x + 1
```