NOVA IMS INFORMATION MANAGEMENT SCHOOL

Programming for Data Science

Lecture 2

Flávio L. Pinheiro

fpinheiro@novaims.unl.pt | www.flaviolpp.com
www.linkedin.com/in/flaviolpp | X @flavio_lpp

Quizzes

https://www.socrative.com

Login as a student

Room Name: **PDS2025**

Student ID: **Student Number**

or

https://api.socrative.com/rc/5NpKRX

Student ID: **Student Number**

Please Join Now!

Lecture 2

- Language Semantics
- Python Data Types and Structures
- Operators
- Flow Control
- Comprehensions, Slices, Typecast

Lecture 2

- **Language Semantics**
- Python Data Types and Structures
- Operators
- Flow Control
- Comprehensions, Slices, Typecast

Semantics

Indentations, not braces!

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl.

```
for i in range(100):
  if i % 2 == 0:
    print("even")
  else:
    print("odd")
```

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block.

Semantics

Indentations, not braces!

end of statements!

Python statements also do not need to be terminated by semi-colons. Semicolons can be used, however, to separate multiple statements on a single line

```
a = 5; b = 6; c = 7
```

```
a = 5
b = 6
c = 7
```

Semantics

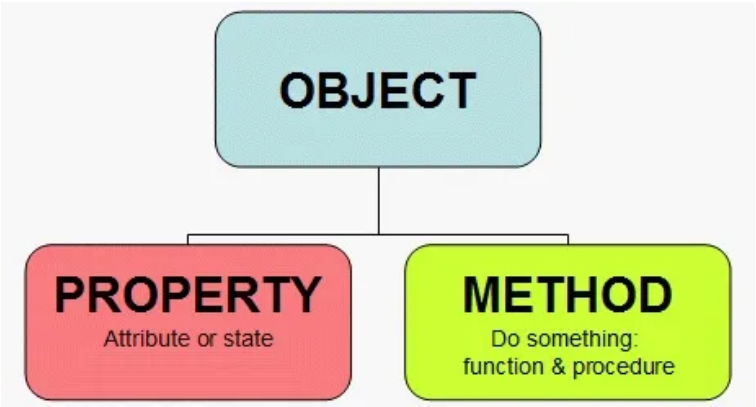
Indentations, not braces!

end of statements!

everything is an object!

An important characteristic of the Python language is the consistency of its object model. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own "box," which is referred to as a Python object.

Each object can have attached variables (attributes) or functions (methods) that allow to manipulate or describe the object instance they belong to!



Semantics

Indentations, not braces!

end of statements!

everything is an object!

hash for comments!

Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want toexclude certain blocks of code without deleting them. An easy solution is to comment out the code:

```
results = []
for line in file_handle:
    #keep the empty lines for now
    #if len(line) == 0:
    #   continue
    results.append(line.replace('foo','bar'))
```

Comments can also occur after a line of executed code. While some programmers prefer comments to be placed in the line preceding a particular line of code, this can be useful at times:

```
print('Reached this line') # status report
```

Lecture 2

- ~~Language Semantics~~
- **Python Data Types and Structures**
- Operators
- Flow Control
- Comprehensions, Slices, Typecast

Variables and Strings

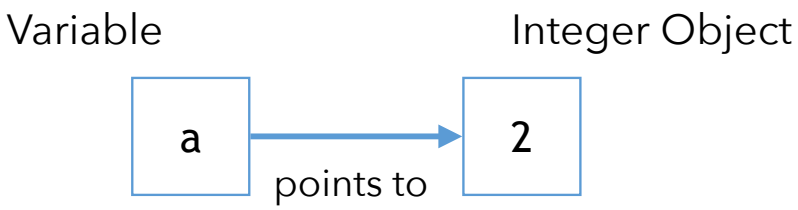
```
>>> a = 2          Declaration of Variable
>>> print(a)
2
>>> type(a)        Check the type assigned
<class 'int'>
```

a is assigned the integer object **2**
which is type **integer**

Variables and Strings

```
>>> a = 2          Declaration of Variable
>>> print(a)
2
>>> type(a)        Check the type assigned
<class 'int'>
```

a is assigned the integer object **2** which is type **integer**



A **Python variable** is a symbolic name that is a reference or pointer to an object. Once an object is assigned to a variable, you can refer to the object by that name. But the data itself is still contained within the object.

Variables and Strings

Declaration of Variable

```
>>> a = 2
>>> print(a)
2
>>> type(a)
<class 'int'>
```

```
>>> a = 2
>>> b = 2.5
>>> c = "Hi there!"
>>> d = 2+3j
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
>>> type(d)
<class 'complex'>
```

Variables and Strings

Declaration of Variable

```
>>> a = 2
>>> print(a)
2
>>> type(a)
<class 'int'>
```

```
>>> a = 2
>>> b = 2.5
>>> c = "Hi there!"
>>> d = 2+3j
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
>>> type(d)
<class 'complex'>
```

Integer

Float

String

Complex

Check the type assigned

Variables and Strings

Most Relevant Variable Types

Method	Description	Example
String	A sequence of Characters in quotation marks	“Hello World!”
Int	Integer Number	-5; 5; 10; 2505 …
Float	Floating number	3.2; 584.2; -35.02 …
Complex	A variable with real and imaginary parts	3-2j; -4+2j …
Bool	Boolean	True or False

Collection Data Structures, Sequences of Objects

List

```
a = [1,'a',3,8,6,5]
```

A list is a variable-length, mutable sequence of Python objects.

Data in Lists is represented inside square brackets.

Tuple

```
a = (1,'a',3,8,6,5)
```

A tuple is a fixed-length, immutable sequence of Python objects.

Data in Tuples is represented inside parenthesis.

Dictionary (hashtable)

```
a = {'a':1,'b':2,'c':3,'d':4}
```

It is a flexibly sized collection of key-value pairs, where key and value are Python objects.

A Dictionary is represented inside curly brackets.

Set

```
a = {1,'a',3,8,6,5}
```

A set is an unordered collection of unique elements. You can think of them like dicts (dictionaries), but keys only, no values.

Data in Sets are represented inside curly brackets.

Collection Data Structures, Sequences of Objects

List

```
>>> a = [1,'a',3,8,6,5]
>>> type(a)
<class 'list'>
```

Tuple

```
>>> a = (1,'a',3,8,6,5)
>>> type(a)
<class 'tuple'>
```

Dictionary (hashtable)

```
>>> a = {'a':1,'b':2,'c':3,'d':4}
>>> type(a)
<class 'dict'>
```

Set

```
>>> a = {1,'a',3,8,6,5}
>>> type(a)
<class 'set'>
```

Collection Data Structures

We can change or not
the elements in these data types

sorted by an index or
history of addition

	Mutable	**Immutable**
Ordered	List	Tuple
Unordered	Dictionary	Sets

List Methods

Method	Description
List append()	Add Single Element to The List
List extend()	Add Elements of a List to Another List
List insert()	Inserts Element to The List
List remove()	Removes Element from the List
List index()	returns smallest index of element in list
List count()	returns occurrences of element in a list
List pop()	Removes Element at Given Index
List reverse()	Reverses a List
List sort()	sorts elements of a list
List copy()	Returns Shallow Copy of a List
List clear()	Removes all Items from the List

Method	Description
list()	Creates a List

Method	Description
any()	Checks if any Element of an Iterable is True
all()	returns true when all elements in iterable is true
ascii()	Returns String Containing Printable Representation
bool()	Coverts a Value to Boolean
enumerate()	Returns an Enumerate Object
filter()	constructs iterator from elements which are true
iter()	returns iterator for an object
len()	Returns Length of an Object
max()	returns largest element
min()	returns smallest element
map()	Applies Function and Returns a List
sorted()	returns sorted list from a given iterable
sum()	Add items of an Iterable
zip()	Returns an Iterator of Tuples

Tuple Methods

Method	Description
Tuple count()	returns occurrences of element in a tuple
Tuple index()	returns smallest index of element in tuple

Method	Description
tuple()	Creates a Tuple

Method	Description
any()	Checks if any Element of an Iterable is True
all()	returns true when all elements in iterable is true
ascii()	Returns String Containing Printable Representation
bool()	Coverts a Value to Boolean
enumerate()	Returns an Enumerate Object
filter()	constructs iterator from elements which are true
iter()	returns iterator for an object
len()	Returns Length of an Object
max()	returns largest element
min()	returns smallest element
map()	Applies Function and Returns a List
sorted()	returns sorted list from a given iterable
sum()	Add items of an Iterable
zip()	Returns an Iterator of Tuples

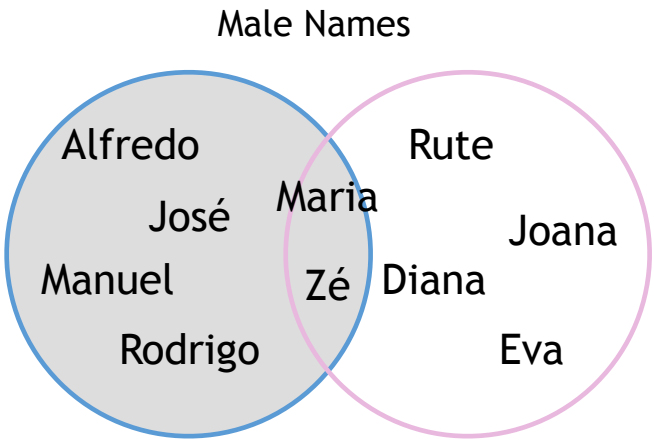
Set Methods

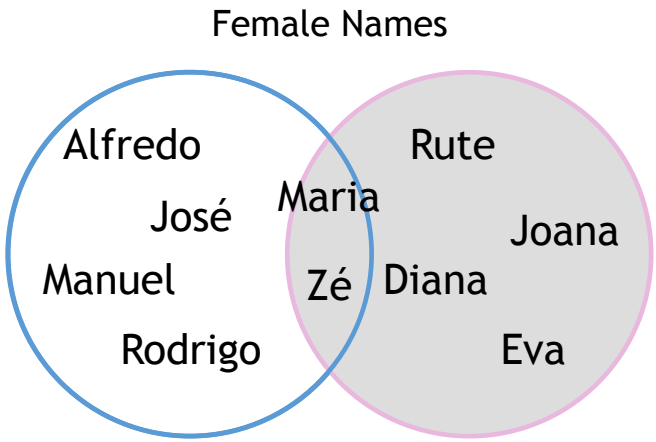
Method	Description
Set remove()	Removes Element from the Set
Set add()	adds element to a set
Set copy()	Returns Shallow Copy of a Set
Set clear()	remove all elements from a set
Set difference()	Returns Difference of Two Sets
Set difference_update()	Updates Calling Set With Intersection of Sets
Set discard()	Removes an Element from The Set
Set intersection()	Returns Intersection of Two or More Sets
Set intersection_update()	Updates Calling Set With Intersection of Sets
Set isdisjoint()	Checks Disjoint Sets
Set issubset()	Checks if a Set is Subset of Another Set
Set pop()	Removes an Arbitrary Element
Set symmetric_difference()	Returns Symmetric Difference
Set symmetric_difference_update()	Updates Set With Symmetric Difference
Set union()	Returns Union of Sets
Set update()	Add Elements to The Set.

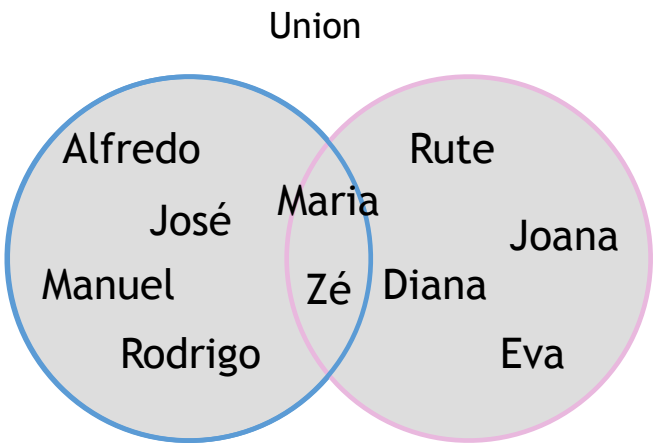
Method	Description
set()	Creates a Set

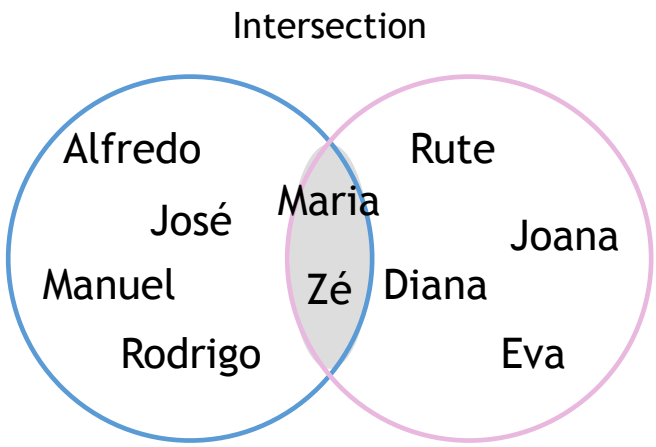
Method	Description
any()	Checks if any Element of an Iterable is True
all()	returns true when all elements in iterable is true
ascii()	Returns String Containing Printable Representation
bool()	Coverts a Value to Boolean
enumerate()	Returns an Enumerate Object
filter()	constructs iterator from elements which are true
iter()	returns iterator for an object
len()	Returns Length of an Object
max()	returns largest element
min()	returns smallest element
map()	Applies Function and Returns a List
sorted()	returns sorted list from a given iterable
sum()	Add items of an Iterable
zip()	Returns an Iterator of Tuples

Set Methods


Male Names
Alfredo
José
Manuel
Rodrigo
Maria
Zé
Rute
Joana
Diana
Eva


Female Names
Alfredo
José
Manuel
Rodrigo
Maria
Zé
Rute
Joana
Diana
Eva


Union
Alfredo
José
Manuel
Rodrigo
Maria
Zé
Rute
Joana
Diana
Eva


Intersection
Alfredo
José
Manuel
Rodrigo
Maria
Zé
Rute
Joana
Diana
Eva

Dictionary Methods

Method	Description
Dictionary clear()	Removes all Items
Dictionary copy()	Returns Shallow Copy of a Dictionary
Dictionary fromkeys()	creates dictionary from given sequence
Dictionary get()	Returns Value of The Key
Dictionary items()	returns view of dictionary's (key, value) pair
Dictionary keys()	Returns View Object of All Keys
Dictionary popitem()	Returns & Removes Element From Dictionary
Dictionary setdefault()	Inserts Key With a Value if Key is not Present
Dictionary pop()	removes and returns element having given key
Dictionary values()	returns view of all values in dictionary
Dictionary update()	Updates the Dictionary

Method	Description
dict()	Creates a Dictionary

Method	Description
any()	Checks if any Element of an Iterable is True
all()	returns true when all elements in iterable is true
ascii()	Returns String Containing Printable Representation
bool()	Coverts a Value to Boolean
enumerate()	Returns an Enumerate Object
filter()	constructs iterator from elements which are true
iter()	returns iterator for an object
len()	Returns Length of an Object
max()	returns largest element
min()	returns smallest element
map()	Applies Function and Returns a List
sorted()	returns sorted list from a given iterable
sum()	Add items of an Iterable
zip()	Returns an Iterator of Tuples

Examples

```
>>> a = [2, 8, 5, 0, 4, 3, 9, 7, 1, 6]
>>> a.sort()
>>> print(a)
[0,1,2,3,4,5,6,7,8,9]
>>> a.append(10)
>>> a.sort(reverse = True)
>>> print(a)
[10,9,8,7,6,5,4,3,2,1,0]
```

Examples

```
>>> a = [2, 8, 5, 0, 4, 3, 9, 7, 1, 6]
>>> a.sort()
>>> print(a)
[0,1,2,3,4,5,6,7,8,9]
>>> a.append(10)
>>> a.sort(reverse = True)
>>> print(a)
[10,9,8,7,6,5,4,3,2,1,0]
```

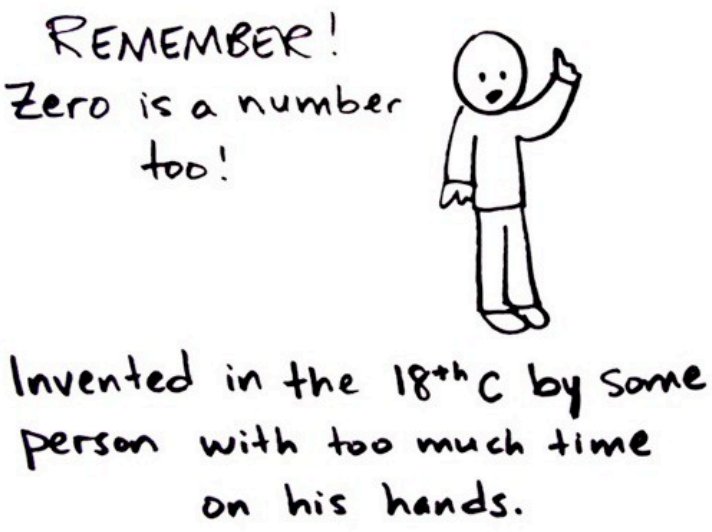
Assigns to **a** a **List** of **Integers**

Sorts the elements of **a** in ascending order

adds one new value at the end of **a**

Sorts the elements of **a** in descending order

Don't Forget Zero



First element of a Collection has index 0!

```
>>> a = [2, 8, 5, 0, 4, 3, 9, 7, 1, 6]
>>> print(a[1])
8
>>> print(a[0])
2
```

a = [2, 8, 5, 0, 4, 3, 9, 7, 1, 6]

0 1 2 3 4 ...

Lecture 2

- ~~**Language Semantics**~~
- ~~**Python Data Types and Structures**~~
- **Operators**
- Flow Control
- Comprehensions, Slices, Typecast

Operators

Operators are special symbols in Python that carry out arithmetic or logical computation between objects. The value that the operator operates on is called the operand.

```
a = 5 + 2
```

Addition Operator

Operators Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Operators Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Operators Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operators Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operators

Comparison

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Operators Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Operators

Identity Operators

Operator	Description
is	Returns True if both variables are the same object
is not	Returns True if both variables are not the same object

Membership Operators

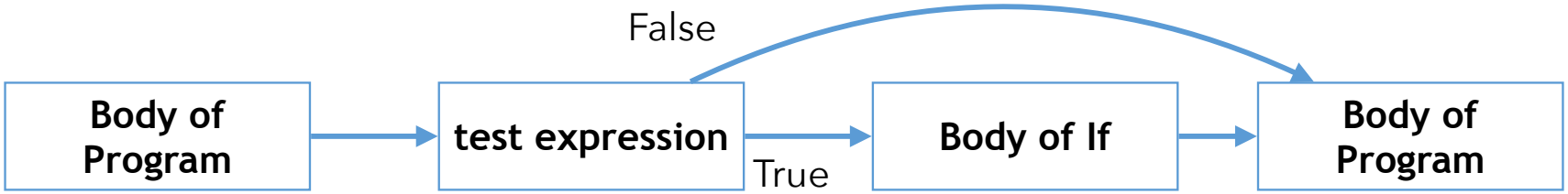
Operator	Description
in	Returns True if a sequence with the specified value is present in the object
not in	Returns True if a sequence with the specified value is not present in the object

Lecture 2

- ~~Language Semantics~~
- ~~Python Data Types and Structures~~
- ~~Operators~~
- **Flow Control**
- Comprehensions, Slices, Typecast

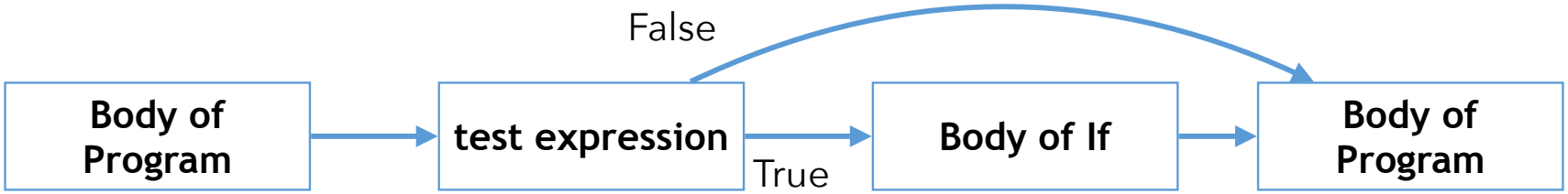
If Statements

an **if** statement allows to perform flow control, creating a condition under which a specific action/task is only executed if a condition (test expression) is true



If Statements

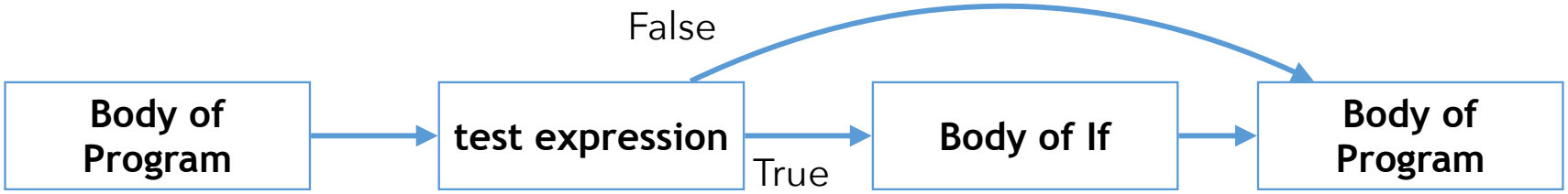
an **if** statement allows to perform flow control, creating a condition under which a specific action/task is only executed if a condition (test expression) is true




```
if x > y:
  %body of statement
  print("x is greater than y")
```

If Statements

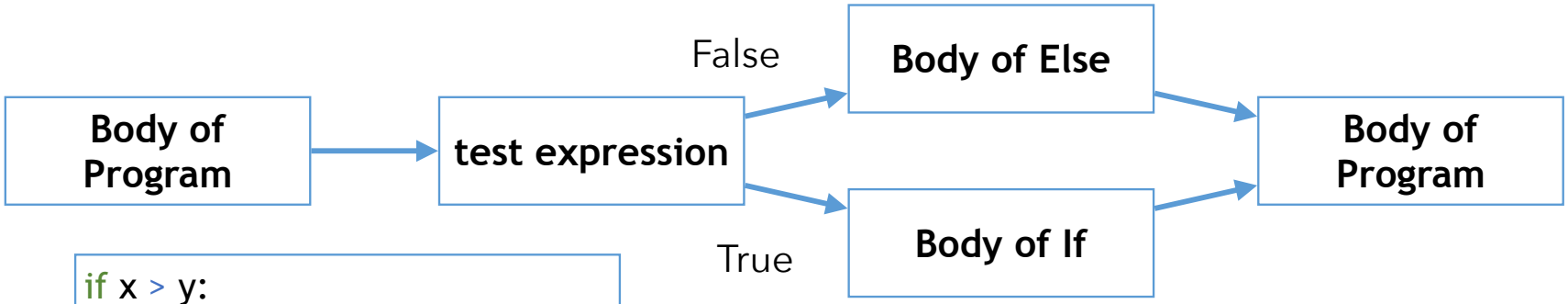
an **if** statement allows to perform flow control, creating a condition under which a specific action/task is only executed if a condition (test expression) is true



Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

If Statements

an **if** statement allows to perform flow control, creating a condition under which a specific action/ task is only executed if a condition (test expression) is true, **else** performs an alternative action




```
if x > y:
    %body of ifstatement
    print("x is greater than y")
else:
    %body of else statement
    print("x is smaller than y")
```

If Statements

if you only have one command to execute
you can save space by writing the **if** and **else** statements inline

with one condition

```
if x > y: print("x is greater than y")
```

```
print("x is greater than y") if x > y
```

with two conditions

```
print("G") if x > y else print ("L")
```

with three conditions

```
print("G") if x > y else print ("E") if x == y else print ("L")
```

If Statements

use the **AND** and **OR** bitwise operators to test for multiple conditions.

with AND

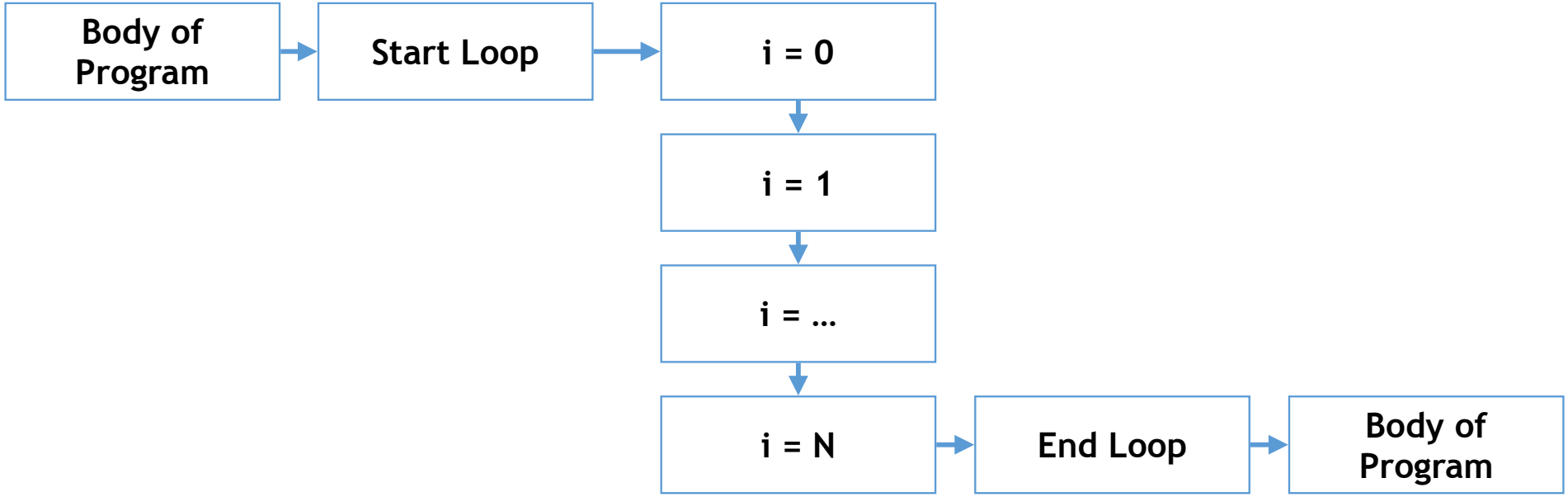
```
if x > y and x > z:
   %body of statement
   print("x is greater than x and z")
```

with OR

```
if x > y or x > z:
   %body of statement
   print("x is greater than x or z")
```

Loops

loop let us **iterate** over a sequence elements or perform a sequence of actions



Loops

The **While** executes repeatedly an action as long as a condition remains true.

```
x = 0
while x < 25:
   print(x)
   x += 1
```

Local condition

increments 1 to x at the end of every iteration

Loops

The **While** executes repeatedly an action as long as a condition remains true.

```
x = 0
while x < 25:
   print(x)
   x += 1
```

```
x = 0
while x < 25:
   print(x)
   if x == 3:
      break
   x += 1
```

break stops the while loop when **x** equals 3

Loops

The **While** executes repeatedly an action as long as a condition remains true.

```
x = 0
while x < 25:
   print(x)
   x += 1
```

```
x = 0
while x < 25:
   print(x)
   if x == 3:
      break
   x += 1
```

```
x = 0
while x < 25:
   x += 1
   if x == 3:
      continue
   print(x)
```

with **continue** we can skip iterations

Loops

The **for** iterates over elements in a sequence (list, tuples, dict, etc)

```
words = ['Banana','Hollywood','Michael']
```

```
for word in words:
   print(word)
```

words in this context is an iterator

Loops

The **for** iterates over elements in a sequence (list, tuples, dict, etc)

```
words = ['Banana','Hollywood','Michael']
```

```
for word in words:
    print(word)
```

```
for word in words:
    if word == 'Banana':
        print(word)
```

```
for word in words:
    if word == 'Banana':
        break
    print(word)
```

```
for word in words:
    if word == 'Banana':
        continue
    print(word)
```

Loops

The **for** iterates over elements in a sequence (list, tuples, dict, etc)

```
for x in range(1,5):
   print(x)
```

```
words = ‘Banana’,’Hollywood’,’Michael’]
for i in range(len(words)):
   print(words[i])
```

length of the list **words**

returns a sequence of numbers starting from 0
and ending at the specified value

What do you expect is going to happen?

```
while 1:
   print(”Where are we going?”)
```

What do you expect is going to happen?

```
while 1:
   print(”Where are we going?”)
```

What is this code doing?

```
>>> a = 5
>>> b = 1
>>> while a > 0:
    ... b *= a
    ... a-= 1
>>> print(b)
120
```

What is this code doing?

```
>>> a = 5
>>> b = 1
>>> while a > 0:
    ... b *= a
    ... a-= 1
>>> print(b)
120
```

computes the factorial of 5

Lecture 2

- ~~Language Semantics~~
- ~~Python Data Types and Structures~~
- ~~Operators~~
- ~~Flow Control~~
- Comprehensions, Slices, Typecast

Comprehensions

Comprehensions

How do we create a list of numbers?

we can create a list, and then fill the list with append() method.

```
lista = []
lista.append(0)
lista.append(1)
lista.append(2)
lista.append(3)
lista.append(4)
print(lista)
```

```
[0, 1, 2, 3, 4]
```

Comprehensions

How do we create a list of numbers?

note that the loops give us some flexibility that append doesn't offer

```
lista = []
for i in range(1,10):
    lista.append(i)
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
list(range(1,10))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
lista = []
i = 0
while i < 10:
    lista.append(i)
    i+=1
print(lista)
```

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

Comprehensions

How do we create a list of numbers?

list with powers of 2

```
lista = []
for i in range(1,10):
    lista.append(i**2)
print(lista)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Fibonacci Sequence

```
lista = [0,1]
i = 2
while i < 10:
    i+=1
    lista.append(lista[i-2] + lista[i-3])
print(lista)
```

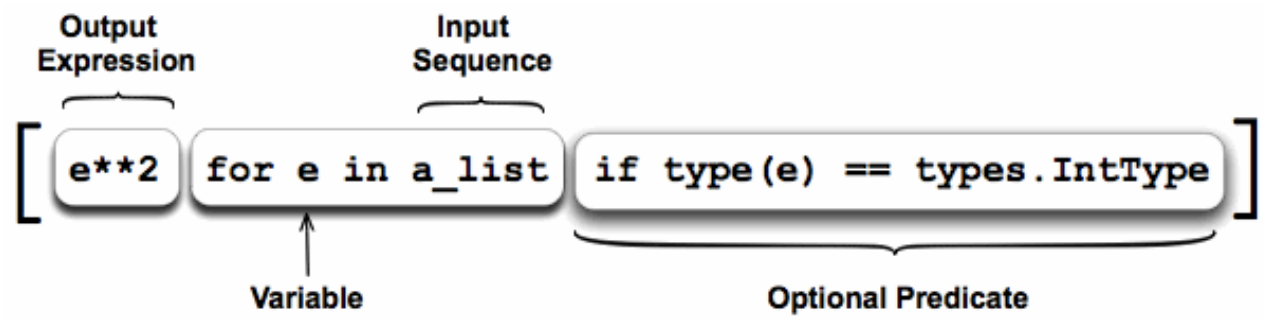
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Comprehensions

Can we save some code?

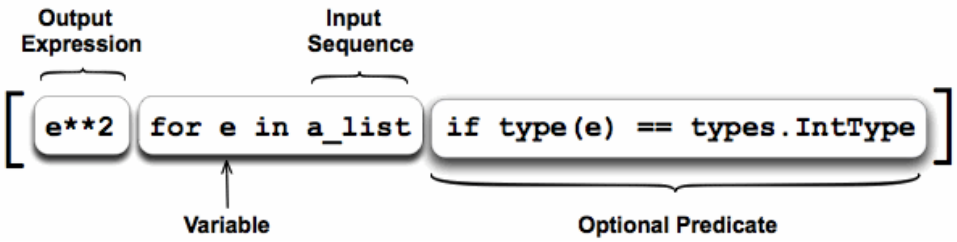
Comprehensions

List comprehensions



Comprehensions

List comprehensions




```
lista = [i for i in range(1,10)]
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
lista = [i**2 for i in range(1,10)]
print(lista)
```

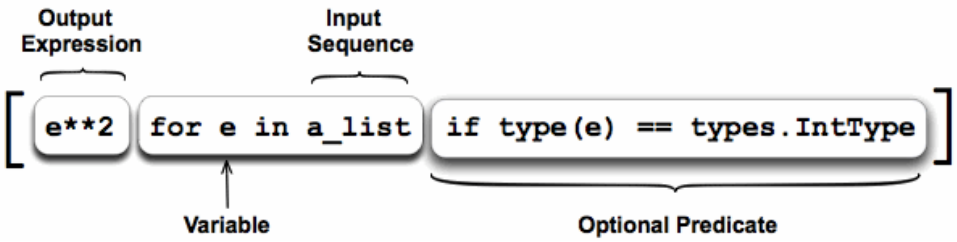
```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
lista = [i for i in range(1,10) if i%2 == 0]
print(lista)
```

```
[2, 4, 6, 8]
```

Comprehensions

List comprehensions



build matrices

```
[ [ row*column for column in range(0, 3) ] for row in range(0, 3) ]
```

```
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

```
[
    [
        1 if column == row else 0
        for column in range(0, 3)
    ]
    for row in range(0, 3)
]
```

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Comprehensions

Comprehensions

dictionaries

```
#{ key_expression : value_expression for expression in iterable }
word = 'letters'
letter_counts = {letter: word.count(letter) for letter in word}
print(letter_counts)
```

```
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

set

```
a_set = {number for number in range(1,6) if number % 3 == 1}
print(a_set)
```

```
{1, 4}
```

tuples don't have a comprehension :sadface

Slices

Slices

Suppose you want to extract a sequence of numbers from a list.

say, from the 2nd to the 5th element in a list, how can you do it?

Slices

Suppose you want to extract a sequence of numbers from a list.

say, from the 2nd to the 5th element in a list, how can you do it?

this does the job,
but it is not very convenient

```
a = [1,2,3,4,5,6,7,8,9,10]
b = []
for i in range(1,6):
    b.append(a[i])
print(a)
print(b)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 3, 4, 5, 6]
```

Slices

Suppose you want to extract a sequence of numbers from a list.

say, from the 2nd to the 5th element in a list, how can you do it?

this does the job,
but it is not very convenient

```
a = [1,2,3,4,5,6,7,8,9,10]
b = []
for i in range(1,6):
    b.append(a[i])
print(a)
print(b)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 3, 4, 5, 6]
```

Is there an easier way?

Slices

Slicing Rules

Slices

Slicing Rules

More about this when we get to Numpy Arrays!

Slices

Suppose you want to extract a sequence of numbers from a list.

say, from the 2nd to the 6th element in a list, how can you do it?

this does the job,
but it is not very convenient

```
a = [1,2,3,4,5,6,7,8,9,10]
b = []
for i in range(1,6):
    b.append(a[i])
print(a)
print(b)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 3, 4, 5, 6]
```

we can slice a instead

```
a[1:6]
```

```
[2, 3, 4, 5, 6]
```

Slices

Slicing

```
a = [1,2,3,4,5,6,7,8,9,10]
```

```
a[1:6]
```

```
[2, 3, 4, 5, 6]
```

Slices

Slicing

```
a = [1,2,3,4,5,6,7,8,9,10]
```

```
a[1:6]
```

```
[2, 3, 4, 5, 6]
```

```
print(a[1:9:2])
```

```
[2, 4, 6, 8]
```

Slices

list[i:j:k]

Negative *i* and *j* are interpreted as *n* + *i* and *n* + *j* where *n* is the number of elements in the corresponding dimension. Negative *k* makes stepping go towards smaller indices.

```
a = [1,2,3,4,5,6,7,8,9,10]
```

```
print(a[-4:-2])
```

```
[7, 8]
```

Typecast

we can transform the type of variable manually,
bellow converting a float to integer

```
a = 99.999
```

```
print(type(a))
```

```
<class 'float'>
```

```
b = int(a)
```

```
print(b)
print(type(b))
```

```
99
<class 'int'>
```

Typecast

we can also transform strings to numbers, but the number in the string needs to be in the right format

```
int("99.9")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-109-bdf6f09a0f77> in <module>()
----> 1 int("99.9")

ValueError: invalid literal for int() with base 10: '99.9'
```

```
int("99")
```

```
99
```

```
float("99.9")
```

```
99.9
```