# Python Programming

Shebna Rose Fabilloren
Philippine Genome Center

## Why Python?

- It is a general purpose language
- Used across the field of biology
- It's free
- Easy to learn

## Topics to be discussed

- Data Types
- Operators
- Variables
- List
- Dictionary
- If-elif-else Statement
- For Loop
- While Loop
- String Operations
- Functions
- Reading and Writing Files

## 2 ways to write Python code

1. Interpreter
   - In your terminal, type “python3”

2. Python script
   - Open text editor
   - Write your code
   - Save as <filename>.py
   - Open terminal
   - Navigate to where you saved your python file
   - Run your python code, “python3 <filename>.py”

## Data Types

- Integer – positive/negative numbers w/out decimal point
- Float – contains decimal point
- String – enclosed in “ ” or ‘ ’
- Boolean – True or False

## Type Casting

- Specify a type on to a variable.

int() - constructs an integer number from an
- integer literal
- float literal (by rounding down to the previous whole number)
- string literal (providing the string represents a whole number)

ex. int(1), int(4.0), int(“5”)

float() - constructs a float number from an
- integer literal
- float literal
- string literal (providing the string represents a float or an integer)

ex. float(1), float(4.0), float(“5.0”)

str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

ex. str(1), str(4.0), str(“5”)

## Arithmetic Operators

- -1
- 4+2
- 4-1
- 4*3
- 2**10
- 11/4
- 0.03 – 0.0009

## Logical Operators

- Not

    - complements the operand

    Ex. not True
- And

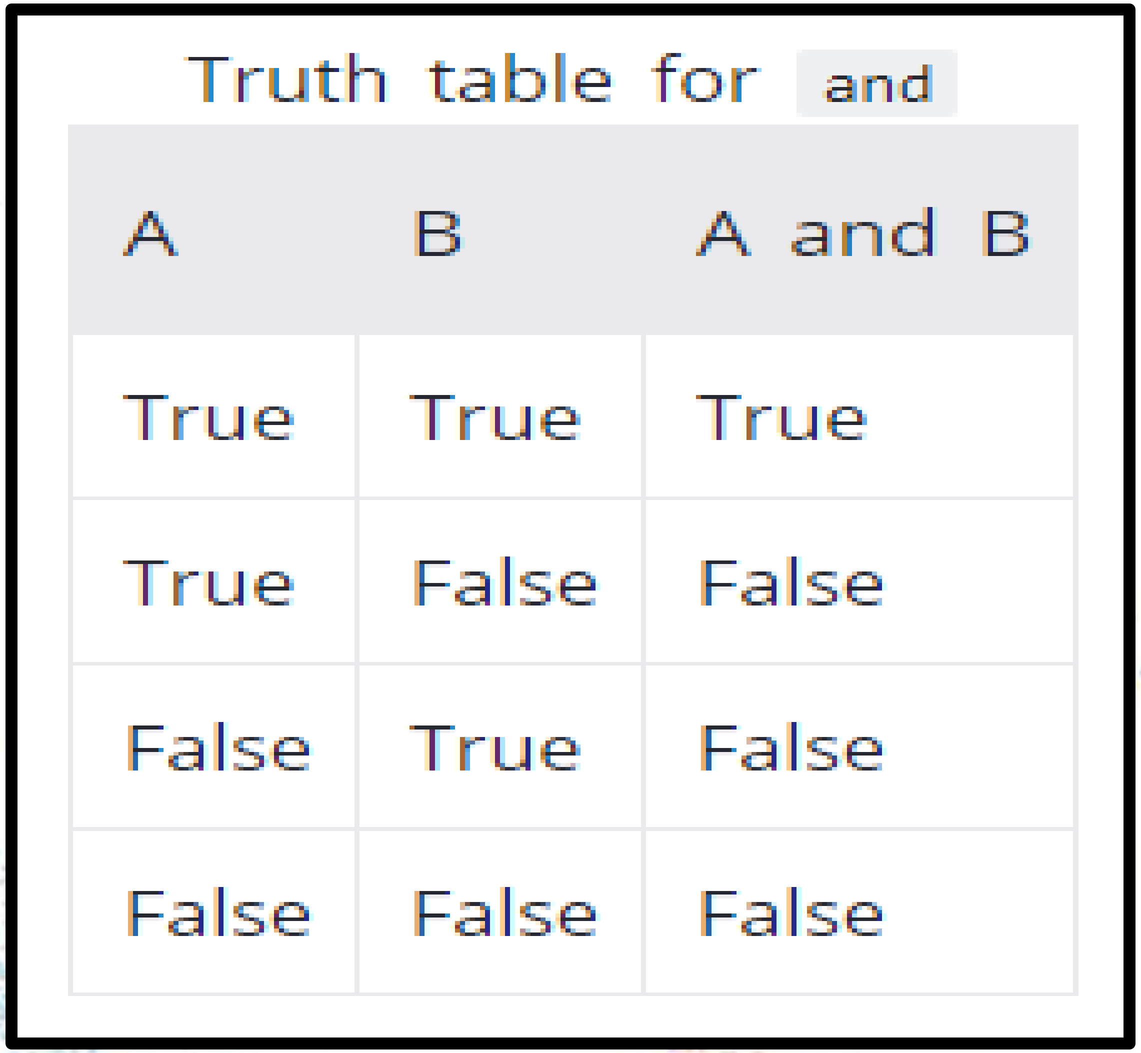
    - True if both operands are true

    Ex. not False and True
- Or

    - True if one of the operands is true

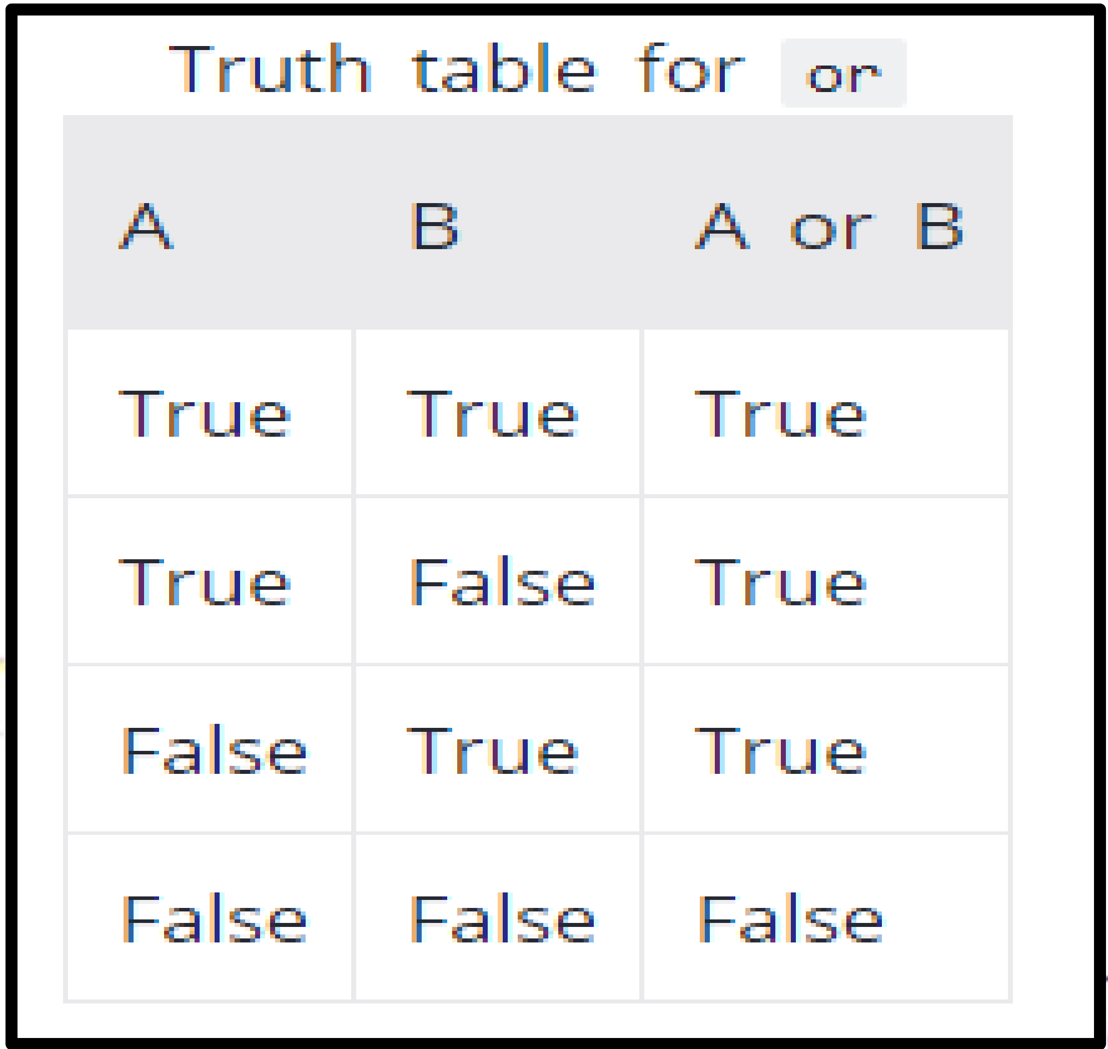
    Ex. Not False or False

## Logical Operators



Truth table for `and`

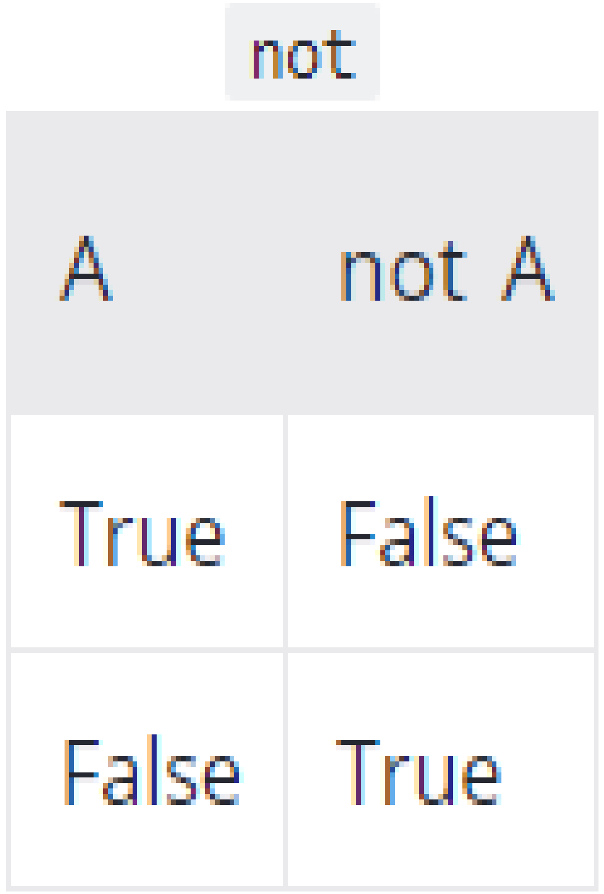
| A | B | A and B |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |



Truth table for `or`

| A | B | A or B |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

## Logical Operators

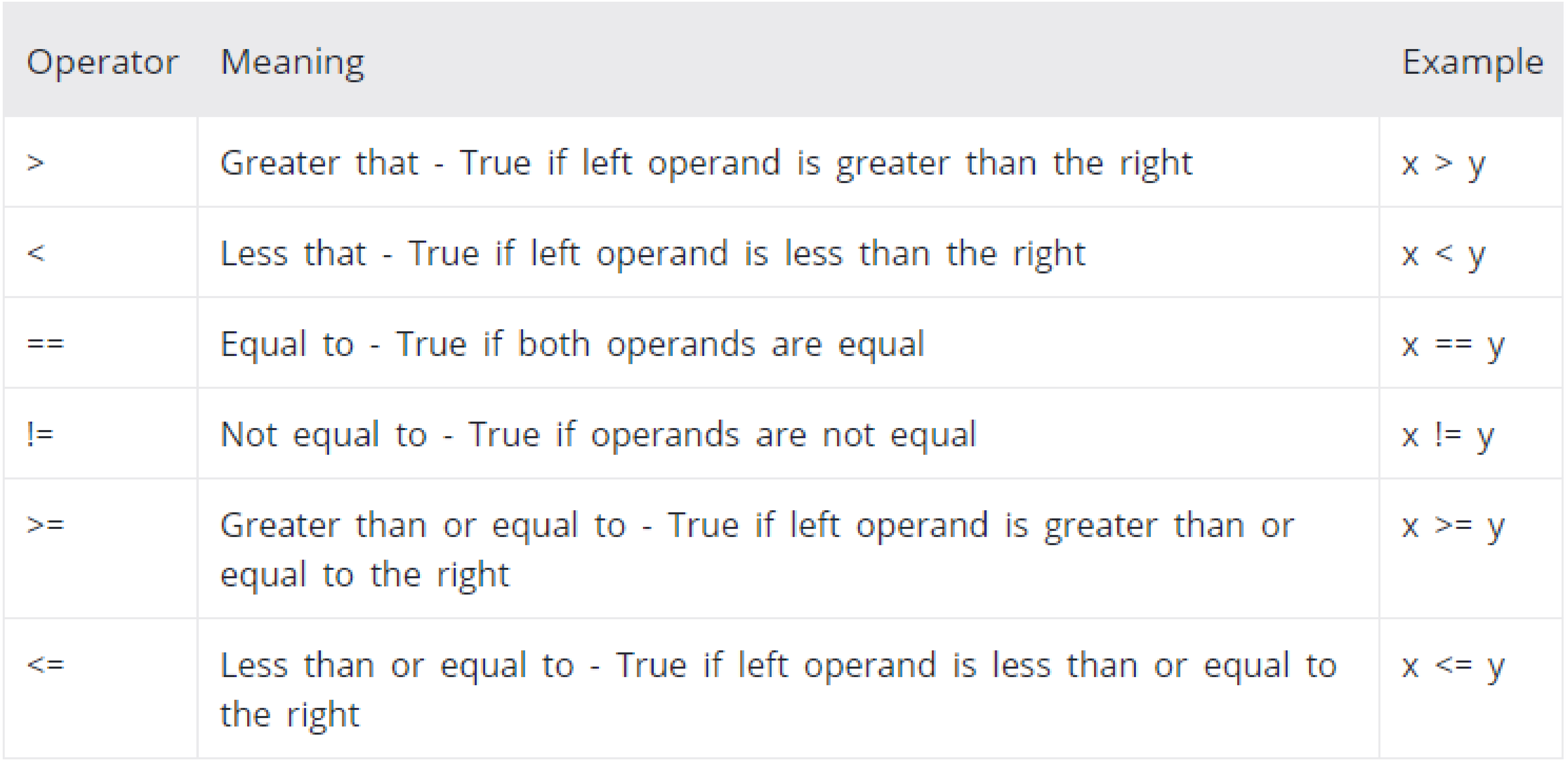


not

| A | not A |
|---|---|
| True | False |
| False | True |

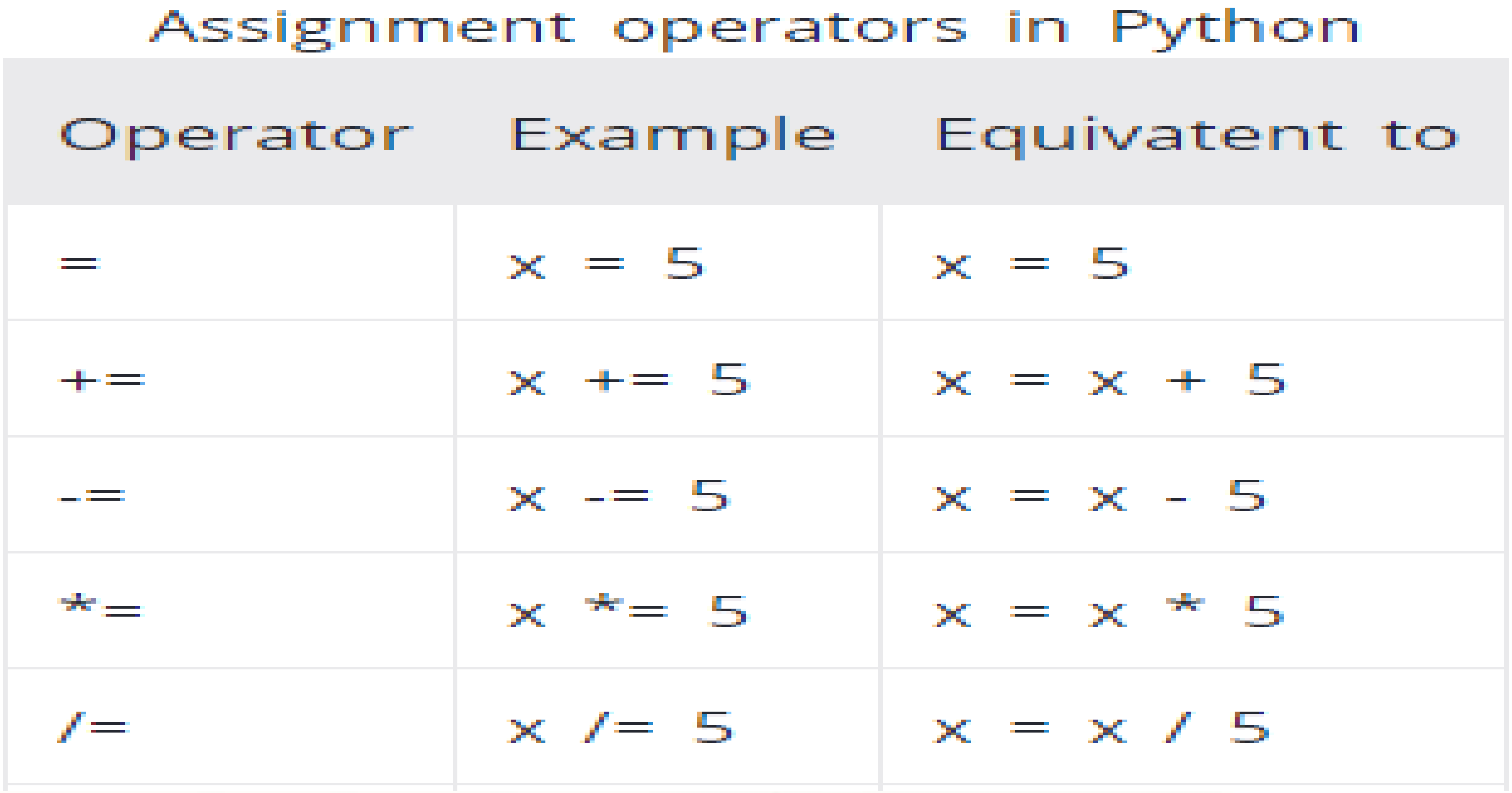
# Comparison Operators

Comparision operators in Python



| Operator | Meaning | Example |
|---|---|---|
| > | Greater that - True if left operand is greater than the right | x > y |
| < | Less that - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

## Assignment Operators



Assignment operators in Python

| Operator | Example | Equivatent to |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| *= | x *= 5 | x = x * 5 |
| /= | x /= 5 | x = x / 5 |

## Variables

- is a named container for data
- think of it as a box or a shelf that has a name
- can hold any type of data
- data in variable can be changed
- text letters, numbers and underscore
- case-sensitive

## Naming Variables

- Reserved keywords
  - Words that can’t be used as variable names because they’re already taken.
  - Ex. if, else, elif, and, or, not, while, break…

## Naming Variables

- Use meaningful variable names
- For multiple words, use camel casing or underscore.
- Make sure it’s readable

## Naming Variables

- Variable names are case sensitive.

myVar is not the same as myvar

## Naming Variables

You can declare multiple variables at once.
Ex.

```
n1, n2, n3 = 1000,2000,3000
```

Note:
Use commas not spaces.

## Example

```
my_DNA = ‘ATGCCGTA’

geneLength = 467

x = False

x = 3.14
```

## Displaying Values

```
print(“I am a string”)

print(87000)

print(6+6)

print(True and False)
```

## Displaying variable values

```
print(variable_name)
```

## Input

- Gets input value from user.

```
print('Enter your name:')
x = input()
Print('Hello, ' + x)
```

## Comments

- Denoted by “#” symbol
- bits of text added by the programmer into the code that explain what is going on.
- not executed by the computer

## List

- Is a collection which is ordered and changeable.
- Position of an element in the list is called **index.**
- Lists start with index 0.
- Represented by elements separated by commas
- Written with square brackets

Example:

```
aList = [1,2,3,4,5]
bList = [1, ‘two’, 3, ‘four’, 5]
```

## List

Accessing Lists:

List elements can be accessed by square brackets [ ].

Syntax:

*list_name*[*index*]

Example:

```
aList = [1,2,3,4,5]
aList[0]
aList[1]
```

## List

Accessing Lists: **Negative Indexing**

- Negative indexing means beginning from the end,
- -1 refers to the last item
- -2 refers to the second last item etc.

Example:

```
aList = [1,2,3,4,5]
aList[-1]
aList[-2]
```

## List

A list can contain another list.
Most often used for dealing with tabular data.

Example:
first_list = [1,2,3,4,5]
other_list = [1, ‘two’, 3, ‘four’, 5]
Nested_list = [1, ‘two’, **first_list**, 4, ‘last’]
[1, ’two’, [1, 2, 3, 4, 5], 4, ’last’]

PHILIPPINE GENOME CENTER

## List

A list can contain another list.
Most often used for dealing with tabular data.

Example:

```
first_list = [1,2,3,4,5]
other_list = [1, ‘two’, 3, ‘four’, 5]
Nested_list = [1, ‘two’, first_list, 4, ‘last’]
[1, ’two’, [1, 2, 3, 4, 5], 4, ’last’]
```

## Multi-dimensional List

Example:

```
a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
print(a)

[[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Access values by:

*list_name*[*row_index*][*column_index*]

|  | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|
| **0** | 2 | 4 | 6 | 8 | 10 |
| **1** | 3 | 6 | 9 | 12 | 15 |
| **2** | 4 | 8 | 12 | 16 | 20 |

## List

Modifying Lists

- Adding, removing, changing the elements

## List

3 Ways of Adding Elements on a List

1. Append
2. Insert
3. Extend

## List

append(element)

- adds an element at the end of the list

```
aList.append(99)
aList
[1,2,3,4,5,99]
```

## List

insert(index, element)

- inserts the element at a specified position.

```
aList.insert(2,50)
aList
[1,2,50,3,4,5,99]
```

## List

extend(list)

- extends a list by adding a list to the end of the original list

```
aList.extend([6,7,8])
aList
[1,2,50,3,4,5,99,6,7,8]
```

## List

Another way of extending a list:

Using the + symbol

```
>>>[1,2,3] + [4,5,6]
[1,2,3,4,5,6]
```

## List

**3 ways to remove elements from a list:**

1. pop([index])
2. remove(element)
3. del

## List

pop([index])

- Removes the element in the index position and returns it to the point where it was called.
- Without parameters, it returns the last element.

1. first_list

```
[1, 2, 50, 3, 4, 5, 99, 6, 7, 8]
```

2. first_list.pop()

```
8
```

3. first_list.pop(2)

```
50
```

4. first_list

```
[1,2,3,4,5,99,6,7]
```

## List

remove(*element*)

- removes the element specified in the parameter
- in the case there's more than one copy of the same element in the list, it removes the *first* one, counting from the *left*.

Example:

```
first_list.remove(99)
first_list
	[1, 2, 3, 4, 5, 6, 7]
```

## List

del list([index])

- has similar effect to list.pop([])

Example:

```
del first_list[0]
[2, 3, 4, 5, 6, 7]
```

## List

Difference between pop() and del?

1. Create 2 empty lists named ‘nucleotides1’ and ‘nucleotides2’
2. Append ‘A’,’C’ to nucleotides1 and ’G’,‘T’ to the nucleotides2.
3. Combine nucleotides1 and nucleotides2
4. Remove the ‘T’ from the list.
5. Remove ‘C’ from the list.
6. Delete the list.

*pop()* returns the extracted element while *del* just deletes it.

## Dictionary

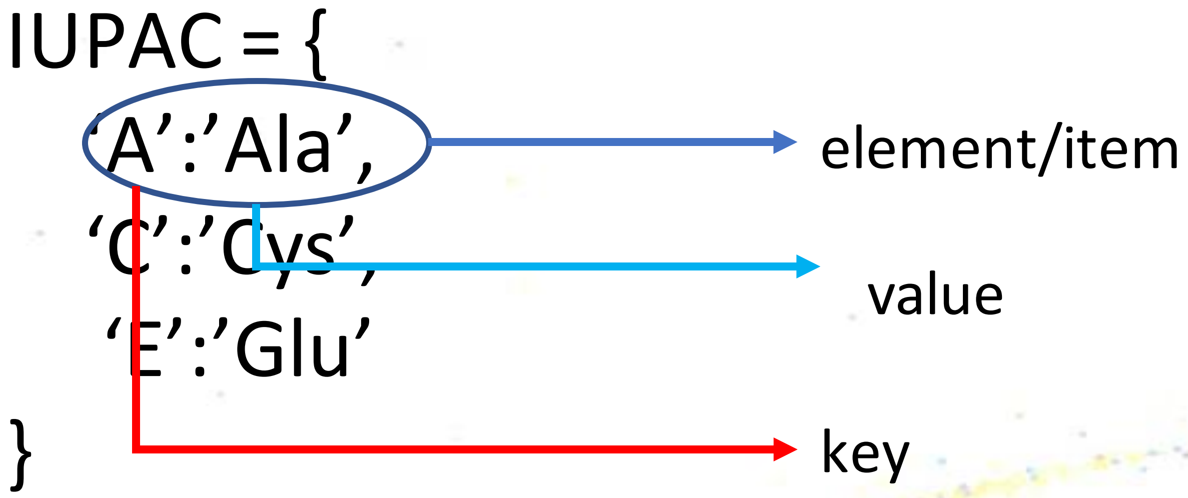
- is a collection which is unordered, changeable and indexed.
- Written with curly brackets
- Has keys and values

## Dictionary

```
dictionary_name = {
    key1: value1,
    key2: value2,
    key3: value3,
    key4: value4
    :
    :
    keyn: valuen
}
```

## Dictionary

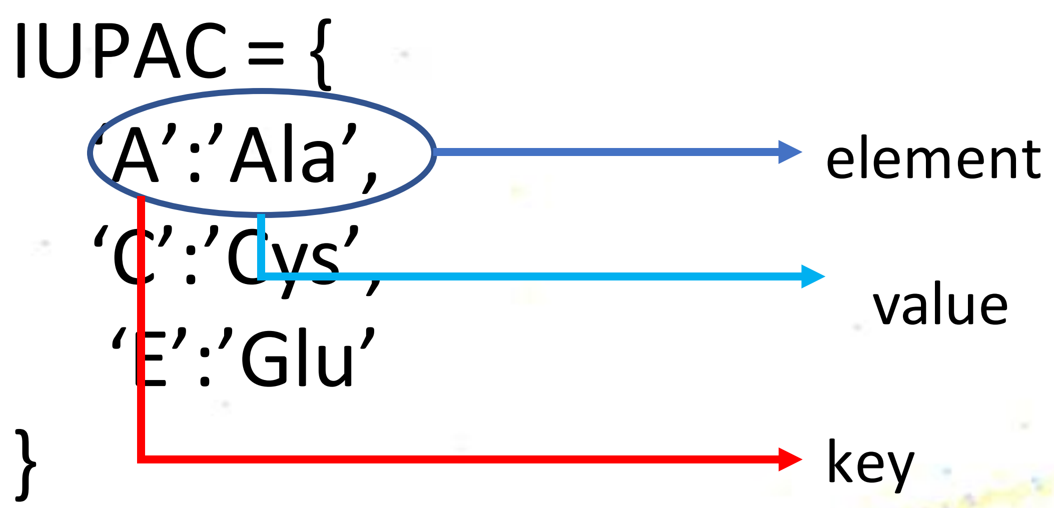
Example:



```
IUPAC = {
    'A':'Ala',
    'C':'Cys',
     'E':'Glu'
}
```

# Dictionary

## Accessing items



```
IUPAC = {
    'A':'Ala',
    'C':'Cys',
     'E':'Glu'
}
```

```
>>>IUPAC['C']
Cys
>>>print('C stands for the amino acid ' +
          IUPAC['C'])
C stands for the amino acid Cys
```

***The key is the index used to retrieve the value.***

## Dictionary

**Changing values**

You can change the value of a specific item by referring to its key name.

```
>>>IUPAC['C'] = Cyster
```

## Dictionary

### Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
>>>IUPAC[‘H’] = His
```

## Dictionary

### Removing Items

The pop() method removes the item with the specified key name:

```
>>>IUPAC.pop('C')
```

# Dictionary

**Delete dictionary**

```
>>>del IUPAC
```

Empty dictionary

```
>>>IUPAC.clear()
```

Kit
Kat

# Flow Control Structures

- managing how and when instructions are executed.

**3 Types**

1. If statements
2. For loop
3. While loop

## IF statements

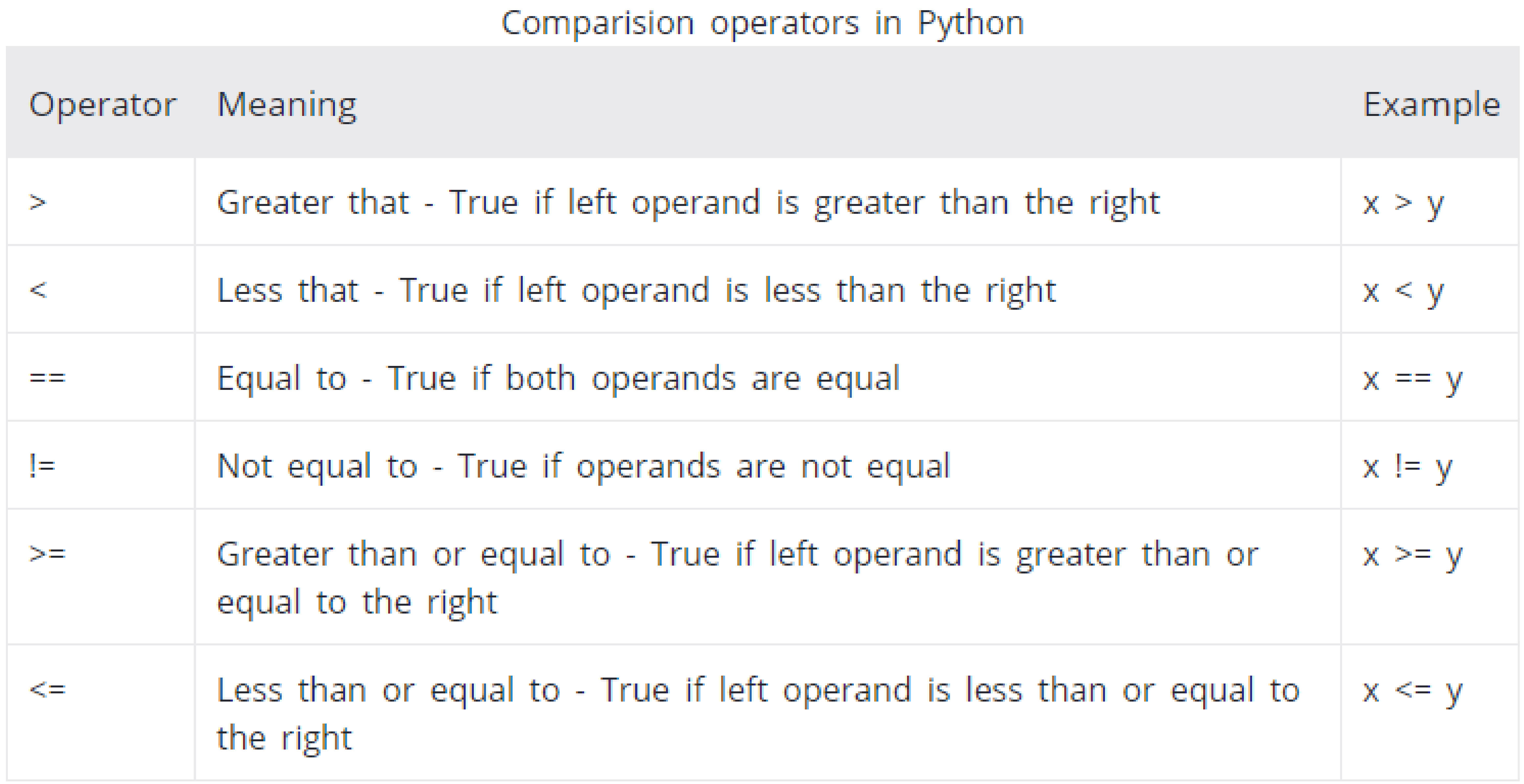
- acts upon the result of an evaluation
- If the expression is true, the block of code just after the if clause is executed.
- Otherwise, the block under else is executed.
- written by the “If” keyword

General Form:

```
if EXPRESSION:
    Block1
else:
    Block2
```

**NOTE:**
**EXPRESSION must return a Boolean value.**

## Comparison Operators



Comparision operators in Python

| Operator | Meaning | Example |
|---|---|---|
| > | Greater that - True if left operand is greater than the right | x > y |
| < | Less that - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

## IF statements

Example:

```
a=8
if a>5:
    print("a is greater than 5")
else:
    print("a is smaller than 5")
```

Program output:

```
a is greater than 5
```

## IF statements

To evaluate more than one condition, use **elif**.

General Form:

```
if EXPRESSION1:
    Block1
elif EXPRESSION2:
    Block2
elif EXPRESSION3:
    Block3
else:
    Block4
```

**Note:**
Once a condition is evaluated as true, the remaining conditions are not checked.

## IF statements

Example using elif

```
dna = input("Enter your DNA sequence: ")
seqsize = len(dna)
if seqsize < 10:
    print("The primer must have at least ten nucleotides")
elif seqsize < 25:
    print("This size is OK")
else:a
    print("The primer is too long")
```

## IF statements

If statements can be nested.

Example:

```
if EXPRESSION:
    if EXPRESSION2:
        Block1
else:
    Block2
```

## IF statements

Example2:

```
dna = input("Enter your DNA sequence: ")
seqsize = len(dna)
if seqsize < 10:
    print("Your primer must have at least ten nucleotides")
    if seqsize==0:
        print("You must enter something!")
elif seqsize < 25:
    print("This size is OK")
else:
    print("Your primer is too long")
```

## For Loop

- Is used for iterating over a list, dictionary, or string.

General Form:

```
for VAR in ITERABLE:
    BLOCK
```

VAR takes the value of the current element of the iterable.

## For Loop

Example:

```
bases = [‘C’,’G’,’T’,’A’]
for x in bases:
    print(x)
```

Program Output:

```
C
T
G
A
```

# For Loop

Example: Using Range

```
y = 0
for x in range(0,10):
    y += x
```

Program Output:

```
45
```

## While Loop

- Similar to the for loop since it also executes a code portion in a repeated way.
- The loop only ends when a given condition is not true.
- There must be an instruction in block to make the *while* condition false. Otherwise, the loop will never end.

General form:

```
while EXPRESSION:
   BLOCK
```

PHILIPPINE GENOME CENTER

## While Loop

Example:

```
a = 10
while a<40:
    print(a)
    a = a+10
```

Program output:

```
10
20
30
```

## While Loop

- Another way to exit a **while** loop is using **break.**
- Loop is broken without evaluating the loop condition.
- **break** is often used in conjunction with a condition that is always true.

**Example:**

```
a = 10
while True:
    if(a<40):
        print(a)
    else:
        break
    a += 10
```

Program Output:

```
10
20
30
```

## String Operations

- In
- Not in
- +
- *
- Subscription
- Slicing

## String Operations

- In
- Not in
- +
- *
- Subscription
- Slicing

## String Operations

In and Not In

- test whether the first string is a substring of the second one (starting at any position). The result is True or False.

Example:

TATA in TATATATATATA

AA in TATATATATA

AA not in TATATATA

## If statement

Example:

```
trans = {"A":"Ala","N":"Asn","D":"Asp","C":"Cys“}
aa = input("Enter one letter: ")
if aa in trans:
   print("The three letter code for "+aa+" is: "+trans[aa])
else:
   print("Sorry, I don’t have it in my dictionary")
```

Output:

```
Enter one letter: A
The three letter code for A is: Ala
```

## String Operations

- In
- Not in
- +
- *
- Subscription
- Slicing

## String Operations

Concatenation

- Join together two strings

Example:

‘AC’ + ‘GT’

‘aaa’ + ‘ccc’ + ‘gg’ + ‘t’

## String Operations

A string can be repeated multiple times.

Example:

```
‘AC’ * 12
6 * ‘TA’
```

## String Operations

- In
- Not in
- +
- *
- Subscription
- Slicing

## String Operations

Subscription

- extracts a one-character substring of a string
- first character is at position 0
- index can be negative
- string[-5]

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[0]

## String Operations

- In
- Not in
- +
- *
- Subscription
- Slicing

## String Operations

Slicing

- extract series of characters from a string
- indices can be positive or negative

```
'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[0:4]

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[:2]
'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[0:]
```

## Defining Functions

The four steps to defining a function in Python are the following:

1. Use the keyword def to declare the function. Follow this up with the function name.

```
def happyBirthday
```

2. Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.

```
def happyBirthday(name):
```

3. Add statements that the functions should execute.

```
def happyBirthday(name):
    print(“Happy Birthday ”,name)
```

4. End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

## Defining Functions

**Function definition**

- subsequent lines are indented relative to the first
- standard practice: indent by four spaces

General form:

```
def function_name(parameter-list):
    function body
```

*What if I have more than one parameter? Separate by comma*

## Using Defined Functions
## Function Call

- write the function name first and enclose within the parentheses your parameters.

```
first_name  = input(“What is your first name? ”)
happyBirthday(first_name)
```

## Return Values

r*eturn* **statement**

- used to return a value from a function

General form:

```
def function_name:
    body
    return value
```

## Return Values

```
def getThirdElement(dna_string):
	third_element = dna_string[2]
	return third_element

myDNA = [“C”,”G”,”T”,”A”]
element = getThirdElement(myDNA)
```

PHILIPPINE GENOME CENTER

## More examples

```
def my_function():
    print(‘Hello from a function’)

def my_function(fname):
    print(‘Hi! My name is ’ + fname)
```

To call a function:

```
my_function()
my_function(fname)

def dog_to_human_years(dog_age):
    new_age = dog_age * 15
    return new_age
```

## Variable Scopes

### Global Scope

```
a = 5
def function():
    print(a)

function()

print(a)
```

## Variable Scopes

### Local Scope

```
a = 5

def function():
    a = 3
    print(a)

function()

print(a)
```

## Variable Scopes

### Implications

```
name = ' Hina'

def change_name(new_name):
    name = new_name

print(name)
change_name('Hokada')
print(name)
```

## File Handling

CRUD (Create, Retrieve, Update, Delete)

open(*“filename”*, *mode*)

- mode:
    a) “r” – Read - Default value. Opens a file for reading, error if the file does not exist
    b) “a” – Append - Default value. Opens a file for reading, error if the file does not exist.
    c) "w" - Write - Opens a file for writing, creates the file if it does not exist
    d) "x" - Create - Creates the specified file, returns an error if the file exist

## File Handling

### Reading Files

**Reading a file**

```
f = open(“myfile.txt”,”r”)
print(f.read())
```

**Reading Only Parts of the File**

```
f = open(“myfile.txt”)
print(f.read(5))
```

**Read Lines**

```
f = open(“myfile.txt”)
print(f.readline())
```

**Looping through the lines of the file (line by line)**

```
f = open("myfile.txt")
for x in f:
  print(x)
```

**Closing the file when you’re done with it**

```
f.close()
```

Close the file to avoid memory usage

## File Handling
Writing to a File

**Writing to an existing file**

Use:
"a" - Append - will append to the end of the file
"w" - Write - will overwrite any existing content

**Append**
Ex.2
```
f = open(“myfile.txt”,”a”)
f.write(“appended text”)
f.close()

#open and read the file after the appending:
f = open("myfile.txt", "r")
print(f.read())
```

**Overwrite**
```
f = open("myfile.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("myfile.txt", "r")
print(f.read())
```

**Create a New File**
```
f = open("myfile.txt", "w")
```

PGC
PHILIPPINE GENOME CENTER

**File Handling**

**Deleting a File**

```
import os
os.remove(“myfile.txt”)
```

# END

PHILIPPINE GENOME CENTER

It's called practice.
C-SPAN
c-span.org