some Basic and Numpy

- To install Python packages, use `py_install()`. For example, we can install the `numpy`, `pandas`, and `matplotlib` packages via R

```
reticulate::py_install(
  packages = c("numpy", "pandas", "matplotlib", "seaborn"
  )
```

Arithmetic Operations

- The arithmetic operations (+, -, *, /) are the exact same

Python

```
x = 11
    x * 2
    x + 2
    x / 2
    x - 2
    x ** 2 # square
    x % 2 # remainder
    x // 2 # Quotient
```

COMMENTs and HELP

- Comments also begin with a #:

Python

```
# This is a comment
```

- Help files are called the same way

Python

```
    help(min)
    ?min
```

Variables

Python is object oriented meaning youu do not need to declare variables before using them, or declare their type The operand to the left of the equal sign is the name of the variable and in the right of the equal sign we have the value stored in the variable.

Python

```
x, variable = "This is a variable", 10
year, v, this_class = 2024, 10.0, "Stat 202"
print(v)
```

```
10.0
```

```
print(this_class)
```

```
Stat 202
```

```
print(year)
```

```
2024
```

How else we can assign a value to a multiple variables?

Python

```
Jim = Sofia = Mike = 19

print(Jim, Jim - Sofia, Mike)
```

```
19 0 19
```

We assign the age of `students` to their name!

Is there any other way?

I think you have seen the following before!

Python

```
John,Alexia,George,State = 22,17,28,"DC"
print("john is", John,
" years old and he is from ", State)
```

```
john is 22  years old and he is from  DC
```

What do you get if you type `print("variable")`?

Python

```
print("variable")
```

```
variable
```

How about `print(variable)`?

Python

```
print(variable)
```

```
10
```

Let define another variable Python

```
z = " and is "
print(z)
```

```
 and is
```

```
print(x)
```

```
This is a variable
```

- What do you think I get with `print(x + z + variable)`?

Python

```
 print(x +z + variable)
```

```
unexpected indent (<string>, line 1)
```

- Since `x`, `z` and `variable` are not all character, or all number then we get error.

How about `print(x, z, variable)`?

Python

```
print(x ,z, variable)
```

```
unexpected indent (<string>, line 1)
```

- We should not get an error here and the result should be "This is a variable and is 10"

What is the type of our Variable, `x`, and `variable`?

Python

```
print(type(x))
```

```
<class 'str'>
```

```
print(type(variable))
```

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

Two main Variable Types

- Number
- String

1. Number.

Python has two types of numbers - integers and floating point numbers(decimals).

Python

```
number_1 = 10
number_2 = 10.5

print("The type of the first number.",
"\n Number ", number_1, " is ", type(number_1))
```

```
The type of the first number.
 Number  10  is  <class 'int'>
```

Python

```
print("The type of the number ", number_2, " is ",
type(number_2))
```

```
The type of the number  10.5  is  <class 'float'>
```

```
print("What is the type of the \"float(10)\"? ")
```

```
What is the type of the "float(10)"?
```

```
print("The type of  \"float(10)\" is ",
type(float(number_1)))
```

```
The type of  "float(10)" is  <class 'float'>
```

```
print("The number_1 is ", number_1,
"\n and the float(number_1) is ", float(number_1))
```

```
The number_1 is  10
 and the float(number_1) is  10.0
```

2. Strings.

Strings are defined either with a single quote or a double quotes.

Python

```
string_1 = 'I am an string'
string_2 = "Me too"

print(string_1 + string_2)
```

```
I am an stringMe too
```

```
print(string_1 , string_2)
```

```
I am an string Me too
```

Do you see the difference between those two?

Question?

What do you get for

a. `print(string_1 + number_1 + string_2)`

Python

```
TypeError: can only concatenate str (not "int") to str
```

b. `print(string_1 ,number_1,  string_2)`

```
I am an string 10 Me too
```

c. `print(number_1 + number_2)`

```
20.5
```

d. `print(Jim + John)`

```
41
```

Lists:

Lists are very similar to arrays. They can contain any type of variable, and they can contain as many variables as you wish.

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Arrays are used to store multiple values of the same type in a single variable.

Python

```
students = ["Jim", 'John', "Sofia"]
print(students)
```

```
['Jim', 'John', 'Sofia']
```

```
s1 = students[1]
print(s1)
```

```
John
```

What is the length of array

Python

```
l = len(students)
print(l)
```

3

```
print(len(s1))
```

4

```
for l in students:
  print(l)
```

```
Jim
John
Sofia
```

Adding an element to an array

use append()

Python

```
students.append("Smith")

print(students)
```

```
['Jim', 'John', 'Sofia', 'Smith']
```

Python

```
students.append(["Susan", "Mike"])

print(students)
```

```
['Jim', 'John', 'Sofia', 'Smith', ['Susan', 'Mike']]
```

Removing array elements

use `pop()`

Python

```
students.pop(4)
```

```
['Susan', 'Mike']
```

```
print(students)
```

```
['Jim', 'John', 'Sofia', 'Smith']
```

- we remove the last element of array

we added again two elements to array. Do you see the difference this time?

Python

```
students.extend(["Susan", "Mike"])
print(students)
```

```
['Jim', 'John', 'Sofia', 'Smith', 'Susan', 'Mike']
```

We can use `remove` method to remove an element

Python

```
students.remove("Mike")
print(students)
```

```
['Jim', 'John', 'Sofia', 'Smith', 'Susan']
```

LISTS

- Python lists are like R lists, in that they can have the different types. You create Python lists with brackets []

Python

```
x = ["hello", 1, True]
x
```

```
['hello', 1, True]
```

BUILT-IN DATA TYPE

There are 4 built-in data types in Python used to store collections of data

1. LIST
2. TUPLE
3. SET
4. DICTIONARY

LIST

Lists are created using square brackets. List items are ordered, changeable, and allow duplicate values Changeable, means we can change, add, and remove items in a list after it has been created.

Python

```
list_1 = students
print(students)
```

```
['Jim', 'John', 'Sofia', 'Smith', 'Susan']
```

Python

```
list_s = ['Sofia', 'Smith', 'Jim',  'John', 'Susan']
print(list_s)
```

```
['Sofia', 'Smith', 'Jim', 'John', 'Susan']
```

```
print("the two list have the same element.",
"\n However, the order is different.",
"\n The two lists are equal \n",
list_1 == list_s)
```

```
the two list have the same element.
 However, the order is different.
 The two lists are equal
 False
```

Python

```
list_2 = [True, False, True, True, True] # Takes Stat class

list_3 = [ 2, 0, 0, 1, 1]  # Number of brothers
```

Python

```
list = [list_1, list_2, list_3]
print(list)
```

```
[['Jim', 'John', 'Sofia', 'Smith', 'Susan'], [True, False,
```

Another way to print

Python

```
list = [list_1, list_2, list_3]
for n in list:
  print(n)
```

```
['Jim', 'John', 'Sofia', 'Smith', 'Susan']
[True, False, True, True, True]
[2, 0, 0, 1, 1]
```

Python

```
list_n = ["stat", 23.6, True]
print(list_n)
```

```
['stat', 23.6, True]
```

Python

```
print(type(students))
```

```
<class 'list'>
```

```
print(type(list))
```

```
<class 'list'>
```

```
print(type(list_1))
```

```
<class 'list'>
```

```
print(type(list_2))
```

```
<class 'list'>
```

```
print(type(list_3))
```

```
<class 'list'>
```

2. TUPLE:

A tuple is a collection which is ordered and unchangeable. Tuples are written with round brackets.

Python

```
tuple_1 = ("GM", "Toyota", "Ford")
```

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created

tuples can have duplicated items

Python

```
tuple_2 = ("Toyota", "Ford", "GM", "Ford")
print(tuple_1)
```

```
('GM', 'Toyota', 'Ford')
```

```
print(tuple_2)
```

```
('Toyota', 'Ford', 'GM', 'Ford')
```

```
print("The two obove tuples are the same: ",
tuple_2 == tuple_1)
```

```
The two obove tuples are the same:  False
```

Python

```
print("The length of tuple_1 is ",
len(tuple_1), " \n and the length of tuple_2 is ",
len(tuple_2) )
```

```
The length of tuple_1 is  3
 and the length of tuple_2 is  4
```

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Python

```
single_tuple = ("A single tuple",)
print("The single tuple is ", single_tuple,
"\n and its type is ", type(single_tuple))
```

```
The single tuple is  ('A single tuple',)
 and its type is  <class 'tuple'>
```

```
print("But without comma we have ", ("A single tuple"),
"\n and its type is ", type("A single tuple"))
```

```
But without comma we have  A single tuple
 and its type is  <class 'str'>
```

A tuple like a list can contain different data types

Python

```
tuple_n = ("stat", 23.6, True)
print(tuple_n)
```

```
('stat', 23.6, True)
```

3. SET

A set is a collection which is unordered, unchangeable*, and unindexed.

Note: Set items are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Python

```
set_1 = {"Toyota", "Ford", "GM"}
print(set_1)
```

```
{'GM', 'Ford', 'Toyota'}
```

Python

```
set_2 = {"GM","Toyota", "Ford",  "Ford"}

print(set_1,
"\n you need to check this with what we actually entered")
```

```
{'GM', 'Ford', 'Toyota'}
 you need to check this with what we actually entered
```

```
print(set_2,
" \n you need to check this with what we actually entered")
```

```
{'GM', 'Ford', 'Toyota'}
 you need to check this with what we actually entered
```

Duplicate value will be ignored

Python

```
set_3 = {"Toyota",  "GM", "Ford"}
print("Set_2 equalls to set_3: ", set_1 == set_3)
```

```
Set_2 equalls to set_3:  True
```

```
print("Set_2 equalls to set_3: ", set_2 == set_3)
```

```
Set_2 equalls to set_3:  True
```

```
print("the type is ", type(set_3))
```

```
the type is  <class 'set'>
```

4. DICTIONARY

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

NOTE:

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Python

```
dict_1 = {
    "name": "Steve McQueen",
    "Known as": "The King of Cool",
    "Born": 1930,
    "Died": 1980
}

print(dict_1)
```

```
{'name': 'Steve McQueen', 'Known as': 'The King of Cool',
```

```
dict_1_a = {
    "name": "Steve McQueen",
    "Born": 1930,
}

print(dict_1_a)
```

```
{'name': 'Steve McQueen', 'Born': 1930}
```

```
dict_1_b = {
    "Known as": "The King of Cool",
    "Died": 1980
}

print(dict_1_b)
```

```
{'Known as': 'The King of Cool', 'Died': 1980}
```

How old was Steve McQueen when he died?

Python

```
print(" He was ", dict_1["Died"] - dict_1["Born"],
" years old")
```

```
 He was  50  years old
```

Python

```
dict_2 = {
    "name": "Steve McQueen",
    "Died": 1980,
    "Known as": "The King of Cool",
    "Born": 1930
}
print(dict_2)
```

```
{'name': 'Steve McQueen', 'Died': 1980, 'Known as': 'The K
```

The two dictionaries are equal

Python

```
print("The two dictionaries are equal ", dict_1 == dict_2,
"\n and the length of them is ", len(dict_1),
"\n Their type is ", type(dict_1))
```

```
The two dictionaries are equal  True
 and the length of them is  4
 Their type is  <class 'dict'>
```

Types of Python Operators

Python

```
op_1 = 10 + 90
op_2 = 10 - 90
op_3 = 10 * 90
op_4 = 13 / 3
op_5 = 86 % 10 # Modules
op_6 = 2 ** 4  # Exponent
op_7 = 13//3  # floor
```

Comparison Operators

Python

```
n = 15
m = 7
print( "n == m", n == m)
```

```
n == m False
```

```
print( "n <= m", n <= m)
```

```
n <= m False
```

```
print( "n < m", n < m)
```

```
n < m False
```

Comparison Operators...

Python

```
print( "n != m", n != m)
```

```
n != m True
```

```
print( "n >= m", n >= m)
```

```
n >= m True
```

```
print( "n > m", n > m)
```

```
n > m True
```

Assignment Operators

Python

```
n+=3
print("We have n = ", n, "then n+=3., makes n = ", n)
```

```
We have n =  18 then n+=3., makes n =  18
```

```
n-=3
print("We have n = ", n,  "then n-=3, makes n = ", n)
```

```
We have n =  15 then n-=3, makes n =  15
```

```
n*=3
print("We have n = ", n,  "then n*=3, makes n = ", n)
```

```
We have n =  45 then n*=3, makes n =  45
```

Assignment Operators...

Python

```
n/=3
print("We have n = ", n,  "then n/=3, makes n = ", n)
```

```
We have n =  15.0 then n/=3, makes n =  15.0
```

```
n%=3
print("We have n = ", n,  "then n%=3, makes n = ", n)
```

```
We have n =  0.0 then n%=3, makes n =  0.0
```

```
n**=2
print("We have n = ", n,  "then n**=2, makes n = ", n)
```

```
We have n =  0.0 then n**=2, makes n =  0.0
```

Floor Division Assignment

Python

```
n //= 3
print("We have n = ", n,  "then n//=3, makes n = ", n)
```

```
We have n =  0.0 then n//=3, makes n =  0.0
```

Bitwise Operators

Python

```
a = 60                # 60 = 0011 1100
b = 13                # 13 = 0000 1101
```

Binary AND

Sets each bit to 1 if both bits are 1 Python

```
c = a & b         # 12 = 0000 1100
print ("a & b : ", c)
```

```
a & b :  12
```

Binary OR

Sets each bit to 1 if one of two bits is 1 Python

```
c = a | b         # 61 = 0011 1101
print ("a | b : ", c)
```

```
a | b :  61
```

Binary XOR

Sets each bit to 1 if only one of two bits is 1

Python

```
c = a ^ b          # 49 = 0011 0001
print ("a ^ b : ", c)
```

```
a ^ b :  49
```

Binary Ones Complement

Inverts all the bits

Python

```
c = ~a            # -61 = 1100 0011
print ("~a : ", c)
```

```
~a :  -61
```

Binary Left Shift

Shift left by pushing zeros in from the right and let the leftmost bits fall off

Python

```
c = a << 2;         # 240 = 1111 0000
print ("a << 2 : ", c)
```

```
a << 2 :  240
```

Binary Right Shift

Python

```
c = a >> 2;         # 15 = 0000 1111
print ("a >> 2 : ", c)
```

```
a >> 2 :  15
```

Logical Operators

logical AND

Python

```
print(True and True)      # True
```

```
True
```

```
print(True and False)     # False
```

```
False
```

```
# logical OR
print(True or False)      # True
```

```
True
```

```
# logical NOT
print(not True)
```

```
False
```

Membership Operators – in and not in

Python

```
x = "Hello, World!"
print("ello" in x) #Returns true as it exists
```

True

```
print("hello" in x) #Returns false as 'h' is in lowercase
```

False

```
print("World" in x)#Returns true as it exists
```

True

not in

Python

```
x = "Hello, World!"
print("ello" not in x) #Returns false as it exists
```

```
False
```

```
print("hello" not in x) #Returns true as it does not exist
```

```
True
```

```
print("World" not in x) #Returns false as it exists
```

```
False
```

Identity Operators —- is and is not

Python

```
m = 70
n = 70
if ( m is n ):
   print("Result: m and n have same identity")
else:
   print("Result: m and n do not have same identity")
```

```
Result: m and n have same identity
```

CONTROL FLOW:

A control flow is the order in which the program's code executes. (conditional statements, loops, and function calls) There are three types of control structures:

1. Sequential - default mode
2. Selection - used for decisions and branching
3. Repetition - used for looping, i.e., repeating a piece of code multiple times.

1. SEQUENTIAL STATEMENTS:

A set of statement whose process happens in a sequence.

Python

```
a = 2
b = 8
a_cube = a**3
c = b -a_cube
print("The value of b = ", b,
"\n and a_cube = ", a_cube,
"\n These values are the same.",
"\n Becuse their difference is c = "
, c)
```

```
The value of b =  8
 and a_cube =  8
 These values are the same.
 Becuse their difference is c =  0
```

2. SELECTION:

it used for decisions and branching.

if

if ...

Python

```
n = 27
if n % 3 == 0:
   print("n =", n,
   "\n is divisible by 3.",
   "\n Then the sum of its digit is also divisible by 3")
```

```
n = 27
 is divisible by 3.
 Then the sum of its digit is also divisible by 3
```

if - else

if ... else ...

Python

```
m = 26
if m % 3 == 0:
    print("m =", m,
    " is divisible by 3.",
    "\n Then the sum of its digit is also divisible by 3")
else:
    print("m =", m, " is not divisible by 3")
```

```
m = 26  is not divisible by 3
```

Nested if

Python

```
a, b, c = 20, 10, 15
if a > b:
   if a > b:
      print("a = ", a, " and it is larger than b =", b)
   else:
       print("b = ", b," and it is larger than a = ", a)
elif b > c:
    print("b = ", b," and it is larger than c =",  c)
else:
     print("c = ", c," and it is larger than b = ", b)
```

```
a =  20  and it is larger than b = 10
```

if-elif-else

Python

```
x = 15
y = 12
if x == y:
   print("Both are Equal")
elif x > y:
    print("x is greater than y")
else:
    print("x is smaller than y")
```

```
x is greater than y
```

INDENTATION

Indentation is very important in python.

It is used to define the blocks of statements.

Python

```
x = 15
y = 12
if x == y:
   print("Both are Equal")
elif x > y:
   print("x is greater than y")
else:
   print("x is smaller than y")
```

```
x is greater than y
```

```
print("pay attention to indentation",
" on the \"if\" statements.")
```

```
pay attention to indentation  on the "if" statements.
```

3. REPETITION, LOOP

A repetition statement is used to repeat a group(block) of programming instructions.

Note: Pay attention to INDENTATION!

FOR LOOP

A for loop is used to iterate over a sequence that is either a list, tuple, dictionary, or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.

Example 1:

Python

```
lst = [10, 20, 30]
for i in range(len(lst)):
   print(lst[i], end=" \n")
```

```
10
20
30
```

- `rang()` function: `range(start, stop, step)`

Example 2:

Python

```
for j in range(10, 15):
    print(j, end=" \n")
```

```
10
11
12
13
14
```

Example 3:

Python

```
for j in range(10, 35, 5):
    print(j, end=" \n")
```

10
15
20
25
30

Note: By default the end key in print is set to new line.

Example 4:

Python

```
for j in range(10, 35, 5):
    print(j, end="\t")
```

```
10	15	20	25	30
```

break STATEMENT:

Immediately exits from the loop and skips remaining expressions in code block

Python

```
for j in range(10, 35, 5):
    i = j -1
    if i >= 16:
        break
print("i = ", i, "and j = ", j)
```

```
i =  19 and j =  20
```

WHILE LOOP

while loops are used to execute a block of statements repeatedly until a given condition is satisfied. Then, the expression is checked again and, if it is still true, the body is executed again. This continues until the expression becomes false.

Python

```
m, i = 25, 10
while i < m:
      print(i, end = " ")
      i += 2
```

```
10 12 14 16 18 20 22 24
```

```
print("End")
```

```
End
```

Print End to signify the end of the program after the loop is completed.

What is different and similarity between for and while loop?

1. For loops know the number of iteration, whereas it is not set for while loop (the number of iteration is unbounded)
2. Both can be ended early by break.
3. For uses a counter,while is also used a counter but it must be initialized before the loop and it must be incremented inside of the loop.

Can rewrite a for loop using a while loop, but it is possible that we can't rewrite a while loop using a for loop.

Python Overview

In R I Want	In Python I Use
Base R	numpy
dplyr/tidyr	pandas
ggplot2	matplotlib/seaborn

Import {numpy} package

Python: `import <package> as <alias>`.

- Let's import the numpy package:

Python

```
import numpy as np
```

- You can use the alias that you define in place of the package name. In Python we write down the package name a lot, so it is nice for it to be short.

- NumPy Arrays are the Python equivalent to R vectors (where each element is the same type). You use the `array()` method of the numpy package to create a numpy array (note that you give it a list as input)

Python

```
vec = np.array([2, 3, 5, 1])
vec
```

```
array([2, 3, 5, 1])
```

- You can do vectorized operations on NumPy arrays

Python

```
vec + 2
vec - 2
vec * 2
vec / 2
2 / vec
```

- Two vectors of the same size can be added/subtracted/multiplied/divided:

Python

```
x = np.array([1, 2, 3, 4])
y = np.array([5, 6, 7, 8])
x + y
x - y
x / y
x * y
x ** y
```

- You extract individual elements just like in R, using brackets []

Python

```
vec[0]
vec[0:2]
```

- Extract arbitrary elements by passing an index array:

Python

```
ind = np.array([0, 2])
vec[ind]
# or
vec[np.array([0, 2])]
```

- **Key Difference: Python starts counting from 0**, not 1. So the first element of a vector is `vec[0]`, **not** `vec[1]`.

- Combine two arrays via `np.concatenate()` (notice the use of brackets here)

Python

```
x = np.array([1, 2, 3, 4])
y = np.array([5, 6, 7, 8])
np.concatenate([x, y])
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Useful functions over vectors

- In R, we have functions operate on objects (e.g. `log(x)`, `sort(x)`, etc).
- Python also has functions that operate on objects. But objects usually have functions associated with them **directly**. You access these functions by a period after the object name. These functions are called "methods". Use tab completion to scroll through the available methods of an object.

Python

```
vec.sort() # sort
vec.min() # minimum
vec.max() # maximum
vec.mean() # mean
vec.sum() # sum
vec.var() # variance
```

More Useful Functions

- But there are still loads of useful functions that operate on objects.

Python

```
np.sort(vec)
np.min(vec)
np.max(vec)
np.mean(vec)
np.sum(vec)
np.var(vec)
np.size(vec)
np.exp(vec)
np.log(vec)
```

Booleans (Python's logicals)

- Python uses `True` and `False`. It uses the same comparison operators as R

Python

```
vec > 3
vec < 3
vec == 3
vec != 3
vec <= 3
vec >= 3
```

Logical Operators

- The logical operators are: **Key Difference**: “Not” uses a different character.
  - & And
  - | Or
  - ~ Not

Python

```
np.array([True, True, False,
False]) & np.array([True, False, True, False])
```

```
array([ True, False, False, False])
```

```
np.array([True, True, False,
False]) | np.array([True, False, True, False])
```

```
array([ True,  True,  True, False])
```

```
~np.array([True, True, False, False])
```

```
array([False, False,  True,  True])
```

- You subset a vector using Booleans as you would in R

Python

```
vec[vec <= 3]
```

```
array([2, 3, 1])
```

- When you are dealing with single logicals, instead of arrays of logicals, use `and`, `or`, and `not` instead

Python

```
True and False
```

```
False
```

```
True or False
```

```
True
```

```
not True
```

```
False
```

Source