INTRODUCTORY SESSIONS

Introduction to Python

Features

- Uses an interpreter
- Can write programs easily (Fewer lines of code)
- Designed for readability

Setting up Python

- Download Python installer from the official website
- Adding environment variables
- IDLE (Python IDE)

Running Python Interpreter

- Open “Command Prompt” or “Terminal”
- Type “python” and press enter

Running scripts

- Open “Command Prompt” or “Terminal”
- Type “python script_name.py”

Sample programs

- Try the following in Python interpreter

  10 + 20

  5 ** 28

  5268 / 56

Basics

Syntax

- No need of line terminators
- Code blocks are identified by indentation

Variables

Variables

- Can be used to keep data in memory

  x = 10

  number = 10.5

  first_name = “ABC”

- No set data types (Dynamic typing)
- Some rules exist for variable names
- “=“ is the assignment operator
- Variable names are case sensitive

Example code

```
x = 5
y = 8

z = x + y

print(z)
```

“print” is a built in function that can be used to get an output to the console.

```
num1 = 5
num2 = 6

print(num1, num2)

num1 = num2
num2 = 8
print(num1, num2)
```

Variables

- Multiple variables on the same line

  n1, n2 = 1.0, 2

  a = b = c = 5

Data Types

Data Types

- Specifies the nature of data
- Python has many data types
  - Text Type: str
  - Numeric Types: int, float, complex
  - Sequence Types: list, tuple, range
    - Boolean Type: bool

Examples

type(3)

type(2.0)

type(“Hi”)

“type” can be used to find the data type.

Numerical Data Types

- Integers
  - Whole numbers with unlimited length can be represented using “int”
- Floats
  - Numbers with decimal points (fractions or whole numbers)
- Complex
  - Complex numbers

Conversions between Data Types

- “int” function
  - int(3.0)
  - int(“42”)
  - int(“8.5”)
  - int(“Hi”)
- “float” function
  - float(3)
- “str” function
  - str(5.0)

Strings

- Double (“”) or single (‘’) quotation marks can be used to specify strings.

  “ABC”, ‘Ab’, “3.0”, ‘1+2’

- Three single or double quotation marks can be used to specify multiline strings.

  “””How
  are
  you”””

Comments

- Comments can be declared by using “#”
- Will not be executed

- Try the following

```
# Assign a value to x
x = 1.0
```

- String literals can also be used as comments

Booleans

- A data type (“bool”)
- Have only two possible values (“True” or “False”)

```
isNumeric = True
type(isNumeric)
```

Conversion to Boolean

- Most values will be evaluated as True except empty values
- 0, “”, [] are some examples for empty values
- “bool” function can be used for conversion

bool(5)

bool(“a”)

bool(0)

bool(“”)

Comparisons

- Comparison operators can be used to compare data

  8.0 > 4

  5 == 6

  4.4 <= 5

- These will be evaluated to Boolean values

Operators

Operators

- Arithmetic Operators
- Assignment Operators
- Logical Operators
- Identity Operators
- Membership Operators
- Bitwise Operators

Arithmetic Operators

- Can be used for arithmetic operations (Calculations)
  - Addition “+”
  - Subtraction “-”
  - Multiplication “*”
  - Division “/”
  - Modulus “%”
  - Exponentiation “**”
  - Floor Division “//”

Assignment Operators

- Used to assign values to variables
  - “=“
  - “+=“ (x += 3 is same as x = x + 3)
  - “-=“
  - “*=“
  - “/=“
  - “**=“
  - “%=“
  - “//=“

Comparison Operators

- Used to compare values
  - “==“ Equal
  - “!=“ Not equal
  - “>” Greater than
  - “<“ Less than
  - “>=“ Greater or equal
  - “<=“ Less or equal

Logical Operators

- Can be used to combine Boolean expressions
- Same functions as +, ., ~ operators in Boolean algebra or logical AND, OR, NOT
  - and (True only if both expressions are True)
  - or (True if at least on of the statements are True)
  - not (True if the statement is False)

Examples

x = 5

x += 3

(x > 5) or (x == 2)

Conditional Structures

IESL GAMES 2022

INTRODUCTORY SESSIONS

Conditional Structures

- “if” structure
  - “if”, “elif”, “else” keywords are used in if statements
  - Represents taking decisions
  - Can execute different code depending on a set of conditions

- Code blocks are identified using indentation
  - Tabs or spaces
  - Same number of spaces or tabs needs to be used in a single code block

if Structure

```
num = 5.0

if num > 4:
        print(“Greater than 4”)
```

if else Structure

```
num = 5.0

if num > 4:
        print(“Greater than 4”)
else:
        print(“Less or equal to 4”)
```

if elif else Structure

```
marks = 95

if marks > 75:
        print(“A”)
elif marks > 45: #can also have any number of elif conditions
        print(“S”)
else:
        print(“F”)
```

if elif else Execution Flow

- Normally Python programs are executed sequentially
- if elif else structure can be used to alter the execution flow

Nested if statements

- “if” statements can be placed inside an “if” statement (Or inside any code block)

```
x = 53

if x >= 0:
        if x < 50:
                print(“x is less than 50”)
        else:
                print(“x is greater than or equal to 50”)
```

Data Structures

What is a Data Structure?

A data structure is a specialized format for organizing, processing, retrieving and storing data.

- Python In-built Data Structures
  - Lists
  - Tuples
  - Sets
  - Dictionaries
- Abstract Data Structures

List

- Lists are used to store multiple items in a single variable.
- Ex:-

```
list1 = ["apple", "banana", "cherry"]

list2 = [1, 5, 7, 9, 3]

list3 = [True, False, False]

thislist = list(("apple", "banana", "cherry"))
```

Length Of a List

- To determine how many items a list has, use the `len()` function
- Try:-

```
thislist = ["apple", "banana", "cherry"]

print(len(thislist))
```

Access List Items

- List items are indexed and you can access them by referring to the index number
- Try:-

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

- Indexing is starting from 0, not with 1.

Change List Items

- To change the value of a specific item, refer to the index number
- Try:-

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Change List Items

- To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values
- Try:-

```
thislist = ["apple", "banana", "cherry", "orange"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

Add List Items

- To add an item to the end of the list, use the `append()` method
- Try:-

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Add List Items

- To insert a list item at a specified index, use the `insert()` method.
- Try:-

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Remove List Items

- The `remove()` method removes the specified item.
- Try:-

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Remove List Items

- The `pop()` method removes the specified index.
- Try:-

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

Tuple

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered, unchangeable, and allows duplicate values.
- Tuples are written with round brackets
- Try:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Tuple

- It is also possible to use the `tuple()` constructor to make a tuple.
- Try:

```
thistuple = tuple(("apple", "banana", "cherry"))
# note the double round-brackets
print(thistuple)
```

Access Tuples

- You can access tuple items by referring to the index number, inside square brackets.
- Try:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Access Tuples

- You can specify a range of indexes by specifying where to start and where to end the range.
- You can specify a range of indexes by specifying where to start and where to end the range.
- Try:

```
thistuple = ("apple", "banana", "cherry", "orange",
"kiwi", "melon", "mango")
print(thistuple[2:5])
```

Change Tuples

- Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.
- Try:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Set

- Sets are used to store multiple items in a single variable.
- A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.
- Sets cannot have two items with the same value. That means duplicates are not allowed.
- Try:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Access Set Items

- You cannot access items in a set by referring to an index or a key.
- But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.
- Try:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Add Set Items

- Once a set is created, you cannot change its items, but you can add new items.
- To add one item to a set use the `add()` method.
- Try:

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Remove Set Items

- To remove an item in a set, use the `remove()`, or the `discard()` method.
- If the item to remove does not exist, `remove()` will raise an error.
- If the item to remove does not exist, `discard()` will NOT raise an error.
- Try:

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Remove Set Items

- You can also use the `pop()` method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.
- The return value of the `pop()` method is the removed item.
- The `clear()` method empties the set.
- Try:

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Session 02

Dictionary

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values.
- Try:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Access Dictionary Items

- You can access the items of a dictionary by referring to its key name, inside square brackets
- Try:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

- There is also a method called `get()` that will give you the same result.

```
x = thisdict.get("model")
```

Access Dictionary Items

- The `keys()` method will return a list of all the keys in the dictionary.
- The `values()` method will return a list of all the values in the dictionary.
- Try:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict.keys())
print(thisdict.values())
```

Change Dictionary Items

- You can change the value of a specific item by referring to its key name
- Try:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Change Dictionary Items

- The `update()` method will update the dictionary with the items from the given argument.
- The argument must be a dictionary, or an iterable object with key:value pairs.
- Try:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Add Dictionary Items

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

- The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

Remove Dictionary Items

- The `pop()` method removes the item with the specified key name.
- The `popitem()` method removes the last inserted item.
- The `clear()` method empties the dictionary.
- Try:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Conditional Statements

Logical Operations

- Python supports the usual logical conditions from mathematics:
  - Equals: `a == b`
  - Not Equals: `a != b`
  - Less than: `a < b`
  - Less than or equal to: `a <= b`
  - Greater than: `a > b`
  - Greater than or equal to: `a >= b`

if Statements

- Above discussed conditions can be used in several ways, most commonly in "if statements"
- conditions can be used in several ways, most commonly in "if statements"
- Try:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

elif Keyword

- The `elif` keyword is pythons way of saying "if the previous conditions were not true, then try this condition".
- Try:

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

else Keyword

- The `else` keyword catches anything which isn't caught by the preceding conditions.
- Try:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Loops

What is a Loop?

- A loop is **a sequence of instruction s that is continually repeated until a certain condition is reached**

- Python has two primitive loop commands:
  - `while` loops
  - `for` loops


for loop
i
for loop
i
for loop

While Loops

With the `while` loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

For Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)
```

The `for` loop does not require an indexing variable to set beforehand.

Looping through a string

```
for x in "banana":
  print(x)
```

Functions

What is a function?

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

How to create a function?

- In Python a function is defined using the `def` keyword:

```
def my_function(age):
  print(age)
```

Calling a function

To call a function, use the function name followed by parenthesis:

```
def my_function(age):
  print(age)

my_function(23)
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name

```
def my_function(fname):
  print(fname + " Refsnes")


my_function("Emil")

my_function("Tobias")

my_function("Linus")
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)


my_function("Emil", "Chek")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

What will do if we want to pass A large number of arguments?

Arbitrary Arguments, *args

```
def my_function(*kids):
  print("The youngest child is " + kids[2])


my_function("Emil", "Tobias", "Linus")
```

Keyword arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)


my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary keyword arguments

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

```
def my_function(**kid):
  print("His last name is " + kid["lname"])


my_function(fname = "Tobias", lname = "Refsnes",
age=23)
```

Default parameter value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
  print("I am from " + country)


my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a list as an argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)


fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return values

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))

print(my_function(5))

print(my_function(9))
```

The pass Statement

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
def myfunction():
  myfunction()
```

Recursion

Python Modules

What is a module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Create a module

To create a module just save the code you want in a file with the file extension `.py`:

Save this code in a file named `mymodule.py`

```
def greeting(name):
  print("Hello, " + name)
```

Use a module

Now we can use the module we just created, by using the `import` statement:

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

Variables in module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Save this code in the file `mymodule.py`

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Contd …

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule


a = mymodule.person1["age"]

print(a)
```

Naming a module

You can create an alias when you import a module, by using the `as` keyword:

Create an alias for `mymodule` called `mx`:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Python built - in - math functions

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

The `min()` and `max()` functions can be used to find the lowest or highest value in an iterable:

The `abs()` function returns the absolute (positive) value of the specified number:

The `pow(x, y)` function returns the value of x to the power of y (x^y).

```
x = min(5, 10, 25)                  // 5

y = max(5, 10, 25)                  // 25

z = abs(-7.25)                     //7.25

w = pow(4, 3)                       //64
```

Python Math module

Python has also a built-in module called `math`, which extends the list of mathematical functions.

To use it, you must import the `math` module:

```
import math
```

When you have imported the `math` module, you can start using methods and constants of the module.

The `math.sqrt()` method for example, returns the square root of a number:

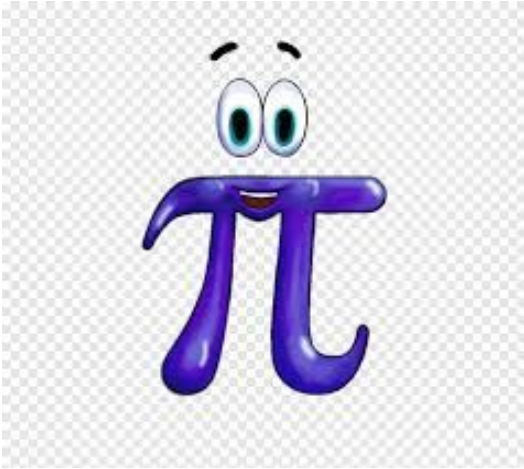
```
import math

x = math.sqrt(64)

print(x)
```

Contd…

The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result:

```
import math

x = math.ceil(1.4)

y = math.floor(1.4)

print(x) # returns 2

print(y) # returns 1
```

Contd ...



The `math.pi` constant, returns the value of PI (3.14...):

```
import math

x = math.pi

print(x)
```

Exception Handling

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `else` block lets you execute code when there is no error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Python User Inputs

Python allows for user input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

```
username = input("Enter username:")

print("Username is: " + username)
```

Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

END

References

- https://www.w3schools.com/python/


Q
&
A

Thank You!!!