Chosen language: **Python**

1. Does the language have static scoping, dynamic scoping, both, or other kind of scoping? Explain.

   Python has **static scoping**. In Python, the bindings between names and objects can be determined at compile time with LEGB rule and keywords: *global, nonlocal* or if not specified those names will be considered as local variables.

```
def n1():
    n = 100
    def n2():
        nonlocal n
        n = 10
    print('before calling n2', n)
    n2()
    print('after calling n2', n)
n1()
```

   Output:

```
before calling n2 100
after calling n2 10
```

2. Does the language have control flow construct? Explain.

   Python **has control flow**:

   Sequencing:

   Python runs statements from the top line to the bottom line. Usually, one line for one statement. Sequence of statements are enclosed as a compound statement using one indentation from the encloser.

```
def encloser ():
    statement1 = 1
    statement2 = 2
    statement3 = 3
    print(statement1 + statement2 + statement3)
```

   Selection:

   Python employs if, elif, ..., else and has short-circuit condition in selection. Thus, the following codes with not yield *IndexError: list index out of range,* because it does only the first condition and short-circuit out.

```
A = [0, 1, 2, 3, 4]
i = 5
if i < len(A) and A[i] > 0:
    print('A[{}] > 0'.format(i))
else:
    print('Duh..')
```

Output:

```
Duh..
```

Python does not have switch-case. Fortunately, one can implement it with Python's dictionary as follows:

```
def case1(*args, **kwargs):
    print('case 1')
def case2(*args, **kwargs):
    print('case 2')
def case3(*args, **kwargs):
    print('case 3')
switch = dict(
    clauseA = case1,
    clauseB = case2,
    clauseC = case3,
)
switch[input()]()
```

This can be done because, Python's Dictionary (or hash table) is equivalent to jump table and functions in Python are considered as objects.

Iteration:

Python has for, while and do-while loops. It also has true iterator for all container abstraction as follows:

```
#true iterator
_list = [1, 2, 3, 4]
for item in _list:
    print(item)
```

Procedural abstraction:

In Python, one can import a library and use its function without having to know how that function works.

```
import math
print(math.sqrt(2))
```

Recursion:

One can create recursion functions like the following:

```
def gcd(a, b):
    if (a == b):
```

```
        return a
    elif (a > b):
        return gcd(a-b, b)
    return gcd(a, b-a)
```

3. Is the language statically typed, dynamically typed, both, or other kind? Explain.

   Python is **dynamically typed**. One doesn't have to explicitly indicate the type of each variable as the following:

```
a = 'string' # string
a = print    # builtin_function_or_method
a('Hello, World')
```

   Python allows *a* to be any type. At the runtime, Python checks if the current type of variable *a* is compatible to doing the method in the statement or not.

4. Does the language use the value model, reference model, both, or other kind of model of variables? Explain.

   Python uses **reference model** as the following:

```
a = 1
print('ref a:', id(a))
b = a
print('ref b:', id(b))
b = 2
print('ref b after:', id(b))
print('value a:', a, 'value b', b)
```

   Output:

```
ref a: 140736645587600
ref b: 140736645587600
ref b after: 140736645587632
value a: 1 value b 2
```

   1 and 2 are immutable. First, *a* points to location of 1. Then, l-value of *a* is dereferenced and l-value of *b* is used. So, *b* is now pointing to 1 just like *a (id(a) and id(b) are the same).* After statement: b = 2, l-value of *b* is changed to the location of 2. Consequently, id(b) is changed and now, expected r-values of *a* and *b* are 1 and 2 respectively.

5. Does the language use type equivalence, type compatibility, both, or other kind of typing rules? Explain.

Python **doesn't directly use type equivalence** since it is dynamically typed and uses reference variable model. However, users can create their own Python default types using C, which uses name equivalence.

Python **uses type compatibility**:

Type Coercion:

Python have type coercion when numeric types (float and int) operate with operands i.e. +, -, * and /. Lower data type will be converted into the higher one to prevent data loss. In additional, Python Class' __method__ is used to emulate desired type. For example, one can create their own numeric type and handle the + operand by:

```
class MyInt:
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        import numbers
        if isinstance(other, numbers.Number):
            return self.num + other
        elif isinstance(other, MyInt):
            return self.num + other.num

i = MyInt(2)
print(i + 0.2)
print(i + i)
```

Output:

```
2.2
4
```

Universal Reference Type:

Every variable is can be used to refer to any type of data as Python uses dynamic typing.

6. Does the language use pass by value, pass by reference, pass by sharing, all of those, or other kind of passing of arguments to subroutines? Explain.

Python uses **pass by sharing**. Parameters are copied and refers to the same location as the original ones. However it can't change the location to which the actual parameters refer.

```
def func2(var): # var refers to location: 1841369997960
    var.append(4) # modified object at var's location. x changes.
    var = [1, 2, 3, 4, 5] # refers to new location. x doesn't change
x = [1, 2, 3] # x refers to location: 1841369997960
func2()
```