DistAlgo is a high-level language for writing distributed algorithms, such that they serve as both clear specifications and runnable implementations of those algorithms. This document provides a brief, practical introduction to the major features of the language using several iterations of a familiar example.

DistAlgo is currently implemented as an extension of the Python language, and requires Python 3.4 or higher. This tutorial assumes some familiarity with Object-Oriented programming in general, and with Python, in particular.

Let's begin with the simplest possible version of "Hello World" written in DistAlgo.

```
 def main():
     print('Hello World.')
```

Listing 1: hello_world01.da - Main Function Definition

Every DistAlgo program must have a main function. In this case, the body of `main` merely contains a call to Python's print function, to output "Hello World" to standard output. As we will see in the next example, the inteded use of the main function in a DistAlgo program is to create, prepare, and then begin the execution of the distinct processes that participate in the distributed algorithm.

In the second version of "Hello World" we can see how process definition and creation works in a DistAlgo program.

```
class P (process):
    def setup(name):
        pass

    def run():
        output('Hello World from:', self.name, ', AKA:', self.id)

def main():
    ps = new(P)
    setup(ps, ('bob',))
    start(ps)
```

Listing 2: hello_world02.da - Process Definition

Distributed algorithms are constituted by the interaction of multiple, distinct processes. DistAlgo is intended for the implementation of distributed algorithms. To that end, DistAlgo makes it easy to define new process types that will execute the behavior required by the algorithm.

Every user defined process type in DistAlgo is an extension of the base class `process`. In the second version of "Hello World" we define a class called `P`, which extends `process`. In order to properly define a new type of DistAlgo process, the user must define two functions: `setup` and `run`.

The `setup` method is used to declare and initialize any instance variables of the user's new process class. Any parameters of the setup method are implicitly declared as instance variables and initialized with the value passed as the argument to the parameter. In the second example, `setup` has one parameter `name`, which is initialized with the value `'bob'`. In addition, every instance of a distalgo process has a unique identifier stored in an instance variable, `self.id`, which is the pair of the IP address and the port number upon which the process instance is listening.

The `run` method of a DistAlgo process contains the definition of the behavior of the process. For the example in listing 2, the process merely prints a "Hello World" message, identifying itself using both the name we supplied as an argument and its unique id.

In this second example we can also see a proper use of the `main` method of a DistAlgo program. Here we have calls to three Distalgo methods: `new`, `setup`, and `start`.

The `new` method is used to instantiate a set of DistAlgo process instances. We supply the classname of the process class we wish to instantiate, `P`, as an argument, and `new` returns a set of references to the new process instances. In this case we store the set in the variable `ps`. If the class name is the only argument, then `new` creates just one instance of that process class. However, even if `new` is used to create just once process instance, the returned value is still a set–in this case a singleton set–containing a reference to that process instance.

The `setup` method is used to initialize our new process instances. The first argument is a set of process instance references. The second argument is a tuple of arguments to be passed to the parameters of the `setup` method defined for the process class corresponding to the class of the process instances one is trying to initialize. Note, even if there is only one argument passed to the `setup` method of the process class, it must still be packaged within a singleton tuple.

Finally, we can begin the execution of the process instances by passing a set of references to those process instances to the `start` method. The call to `start` will cause the `run` method defined within the corresponding process class to begin executing at each process instance included in the call to `start`.

In the third example, we can see how multiple process instances are created.

```
class P (process):
    def setup(name):
        pass

    def run():
        output('Hello World from:', name, ', AKA:', self.id)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    setup(ps, ('bob', ))
    start(ps)
```

Listing 3: hello_world03.da - Multiple Process Instances

You can run this program from the command line by entering: “python3 -m da hello_world3.da”.

This program is very similar to “hello_world2.da”. In fact, the definition of the process class, `P`, is identical to the one given in the previous example. There are just two changes in `main`. First, we add a new variable, `num_processes`, which is initialized to 2. Second, we pass this new variable as the argument to the `num` parameter of the `new` method, resulting in the instantiation of two process instances of the process class, `P`. These two instances are both initialized by the call to `setup`, and then set running by the call to `start`.

The instantiation and execution of the two process instances can be observed in the output when the program is run. There will be two “Hello World” outputs, each with a distinct value for `self.id`, though both will claim that their name is 'bob', since they are both inititialized with the same parameter. You should think about how you might initialize the `name` fields of these two process instances with distinct values.