# Arguments and Parameters:

## Parameter passing by value:

We have said that parameter passing is always by copying the argument to parameter. This is called "**parameter passing by value**". The argument could be a simple or value type like int, float or it could be a structured type or reference type like list, dict. Would these make a difference?

Let us try some simple examples to understand this concept.

```
# name: 1_fn_parameter_passing.py
def fn1(x):
      x = 20

a = 10
fn1(a)
print(a) # 10
#-----------------------------------

def fn2(x):
      x += x

a = 10
fn2(a)
print(a) # 10
#-----------------------------------

def fn3(x):
      x = [33, 44]

a = [11, 22]
fn3(a)
print(a) # [11, 22] ; no change
#-----------------------------------

def fn4(x):
      #x += [33, 44]
      x.append(55)

a = [11, 22]
fn4(a)
print(a) # changes
#-----------------------------------

def fn5(x):
      x[0]=33

a = [11, 22]
fn4(a)
print(a) # Output is [33, 22]
```

```
# modify a reference: others not affected
# modify through a reference: others are affected
```

**Let us check the function fn1.**

```
def fn1(x):
      x = 20

a = 10
fn1(a)
print(a) # 10
```

The function fn1 is created. The variable a becomes 10. The function fn1 is called with the argument a whose value is 10. On the function call, the activation record of fn1 is created with the parameter x. The parameter x gets a copy of the argument a which is 10. The parameter is a local variable of the function fn1. It is changed to 20. When the end of the function is reached, the parameter is never copied to the corresponding argument. So, the variable a of the caller remains unchanged.

**It is not possible to change an argument of a simple type by calling a function.**

**Let us check function fn2.**

```
def fn2(x):
      x += x

a = 10
fn2(a)
print(a) # 10
```

This is similar to fn1. The parameter x which is a copy of the argument a is changed by doubling. The id of x does not change. But the argument is not affected by the function call.

**Let us examine function fn3.**

```
def fn3(x):  # here x is a new reference to same list a
      x = [33, 44] # A new list object is assigned to x

a = [11, 22]
fn3(a)
print(a) # [11, 22] ; no change
```

The argument in this case is a reference type. The argument a is copied to the parameter x. The id of x and id of a will be same. The elements of the list are not copied on this function call. But changing the parameter by assignment will create a new list breaking the relation of x with the list a. But the argument a is not affected.

**Changing the parameter of the reference type does not affect the argument.**

**Let us examine the function fn4.**

```
def fn4(x):
      #x += [33, 44]
      x.append(55)

a = [11, 22]
fn4(a)
print(a) # changes
```

As in the last case, both a and x will refer to the same list as x is a copy of a. When we append to x, we are not modifying x and but we are modifying what x refers to. So a also gets changed.

Please check all these programs on the Python tutor. I suggest that you draw the schematics and understand how the parameter passing works.

### Changing through the parameter will affect the argument.

Let us try to understand the consequence of this parameter passing mechanism.

**(p, q) = (q, p)**

The above statement swaps variables p and q.
But the code below fails to swap the arguments no matter what type they are.
The copies get swapped in the function but not the arguments!

```
#name : 2_fn_swap.py
def swap(p, q) :
      (p, q) = (q, p)

x, y = 11, 22
swap(x, y)
print(x, y) #does not swap

x, y = [11, 22], [33, 44]
swap(x, y)
print(x, y) #does not swap
```

**Moral of this story:** cannot write a function to swap variables in a language like Python which does parameter passing by value.

# Function Parameter and Types:

```
# name : 3_fn_positional_keyword_parameter.py

def fn1(a, b):  # the type of parameters a and b is generic
      print("ok")

fn1(10, 20)
fn1(True, "true")
fn1([11, 22], {33:44})
#fn1(1, 2, 3) # Error
#fn1(4) # Error


def foo(a, b, c, d):
      print(a, b, c, d)


# matching of arguments to parameters:
# 1. positional parameter
#      match the argument to the parameter based on the position
# 2. keyword or named parameter
#     specify the parameter name in the function call
#     match the argument based on the name of the parameter
#     order of arguments does not matter

foo(1, 2, 3, 4)

foo(b = 2, d = 4, a = 1, c = 3)

# specify all the positional parameters before keyword parameters
foo(1, 2, d = 4, c = 3) # c = 3 is not an assignment
#print(c) # error


def fn1(a, b):
      print("ok")
```

Let us examine the above function. What can be the arguments for this function? Should they be of simple type? Can a and b be different types? Can the types be simple or reference types? Answer is yes to all of them.
In the first case, both are int. In the second case, the first is a bool and the second is a str. In the third case, the first is a list and the second is a dict. All these are perfectly fine.

```
fn1(10, 20)
fn1(True, "true")
fn1([11, 22], {33:44})
```

**The parameter has no specified type. The type is decided when the argument is copied on call to the function at runtime – type is determined dynamically.**

What if the number of arguments does not match the parameter?

```
#fn1(1, 2, 3) # Error
#fn1(4) # Error
```

The translator would not know how to match. So both these are errors.

# Matching of arguments to parameters:

### There are 2 - ways of matching arguments to parameters.

```
def foo(a, b, c, d) :
      print(a, b, c, d)
```

### a) Positional parameters.

In this case, we match the argument to the corresponding parameter. We match the first argument to the first parameter, the second argument to the second parameter, and so on from left to right.

```
foo(1, 2, 3, 4)
```

### b) Keyword parameters

In this case, we specify the name of the parameter and then symbol = and then the argument. In this case, **the order of arguments does not have any effect.** The symbol = does not stand for assignment. As we know, assignment is not an expression and no variable is created in the environment of the caller.

```
foo(b = 2, d = 4, a = 1, c = 3)
```

We can also use a combination of these two techniques as long as all the positional parameters have been given the corresponding arguments.

```
foo(1, 2, d = 4, c = 3)
```

This feature makes the names of the parameters also an interface. As long as they have meaningful names, they will be calling the functions simple and less error prone. With default parameters (yet to be discussed), the life of the programmer becomes simple and easy.

# Function Parameters: Default Parameters:

What does print display at the end of the record? What separator do we get between the fields? We know that we get a newline and a space respectively by default. We also know that we can use keyword parameter and specify values for these two parameters in the call to the print function.

How do we specify the default parameter?

```
# name: 4_fn_parameter_default.py

# functions: # default parameter
#      Parameter can be associated with a value
#      if the argument is not provided for this parameter, then default value will be used
#      else default value is not considered
#
#      default value for a parameter is part of the function definition

# 1. simple example

def multiply(a, b = 10):
      return a * b

print(multiply(10, 20))
print(multiply(30))


# 2. default value being a variable

x = 222
# default parameter can be a variable provided it is defined before the function definition
def foo(a = x) :
      print("foo : ", a)

x = 333 # will have no effect on the default stored in the function foo for a
foo(111) #111
foo() #222
```

**The default parameter as part of the function definition. The rightmost parameters can be default**. The function *multiply* can be called with one or two arguments. If the function is called with two arguments, the default parameter does not come into picture. The given argument is copied to b.

If the function is called with one argument, then the default parameter 10 is copied to b before the execution of the function starts.

**The default parameter is processed when the leader is processed and is stored as an attribute of the function definition and is not part of the activation record.**

```
def multiply(a, b = 10):
      return a * b
print(multiply(10, 20))
print(multiply(30)) # multiple(30, 10)
```

**The default can be initialized by any expression which is available at the point the leader of the function is processed.**

```
x = 222
def foo(a = x) :
      print("foo : ", a)
```

**So, in this case, a takes the present value of x which is 222. Change of x later will have no efect on the default parameter stored as an attribute of the function definition.**

## The behavior of the default appears to be unusual if the default parameter is of reference type. Let us examine the next example.

```
# name: 5_fn_parameter_default.py
# parameters : default

def foo(x, a = []):
      a.append(x)
      print(a)

foo(10) #[10]
foo(20) #[10, 20]

z = [30, 40]
foo(50, z)  #[30, 40, 50]
foo(60)     # [10, 20, 60]
```

### def foo(x, a = []) :

In the above leader statement the parameter has a default which is a list – a reference type. That **reference will remain with the function entity**. **The default parameter is copied to a only if no argument is passed for a. if the list is changed, then that change will remain to persist and will manifest the next time the call is made without argument for a.**

Let us examine each call.

**Default parameter is an empty list.**

`foo(10) #[10]`

On this call, the empty list is appended 10. so the default parameter which is part of the function definition is a list containing one element 10

`foo(20) #[10, 20]`

The default parameter which is a list containing one element is copied to the parameter a. One more element 20 gets appended. So the default parameter is a list of two elements 10 and 20

```
z = [30, 40]
foo(50, z) #[30, 40, 50]
```

**The default does not come into picture. It exists, but not used.**

`Foo(60) # [10, 20, 60]`

The default parameter reappears!. So a becomes a list containing two elements to which the third element is appended.

**Do try this example on python tutor.**