Functions

4/19/2022

Content Warning

Fake Violence & Flashing Lights


Replay


Replay

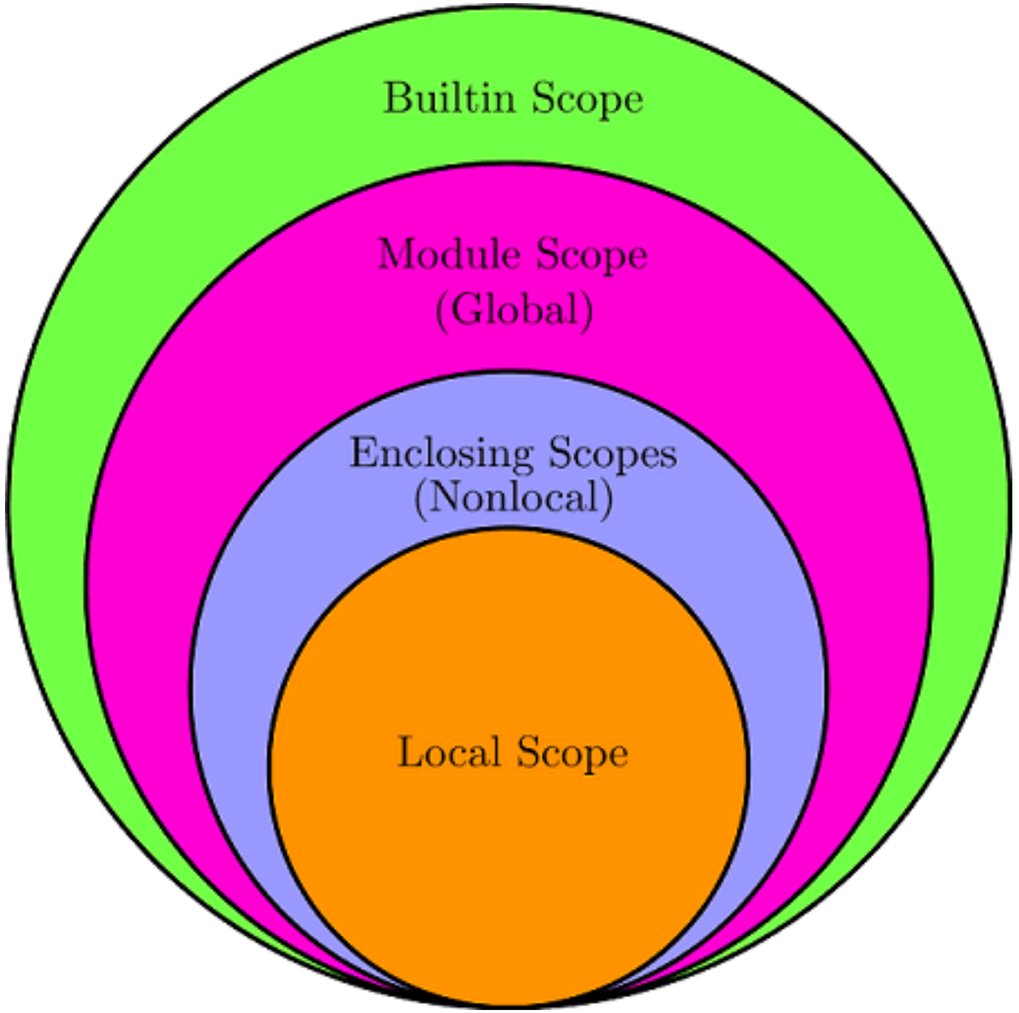
https://drive.google.com/file/d/181FkwUQJvJvm6E29XKO9zap_s6Flj-qR/view

Namespace

- Namespace is how python keeps track of what variable corresponds to what object
- Backed by a python dictionary

Scope

- Scope determines what order we check these namespaces



Scope in action

Scope in action

```
x = 5

def enclosing():

    y = 6

    def local():

        z = x + y
```

Scope in action

We first look for x locally

(In the current function)

Scope in action

We first look for x locally

(In the current function)

```
x = 5

def enclosing():

    y = 6

    def local():

        z = x + y
```

Scope in action

We next look for x in enclosing functions (if there are any)

Scope in action

We next look for x in enclosing functions (if there are any)

```
x = 5

def enclosing():

    y = 6

        def local():

            z = x + y
```

Scope in action

Next, in the global scope

Scope in action

Next, in the global scope

```
x = 5

def enclosing():

    y = 6

    def local():

        z = x + y
```

Scope in action

Success! We have found X.

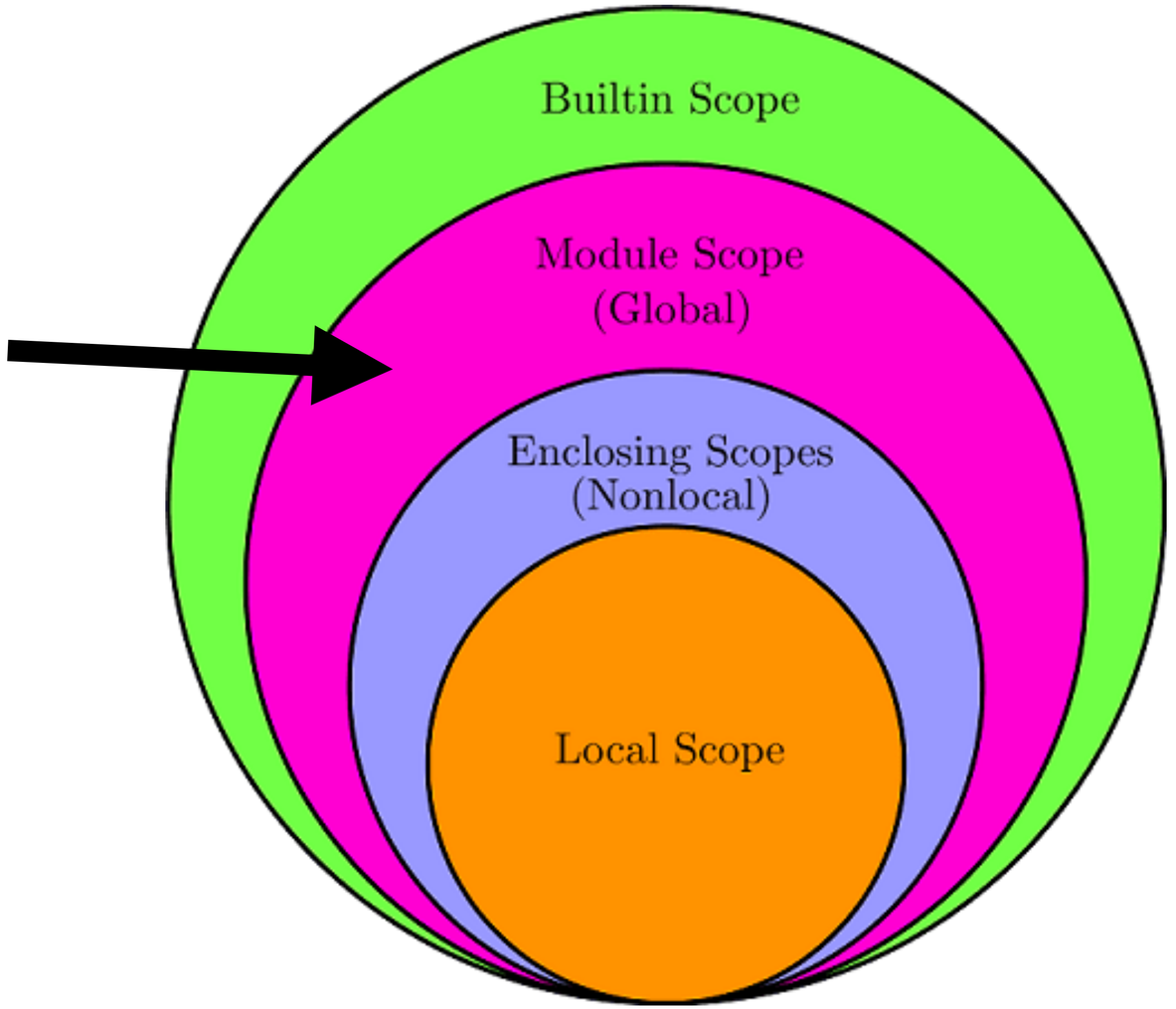
Scope in action

Success! We have found X.

```
x = 5

def enclosing():

    y = 6

    def local():

        z = x + y
```

Scope in action

We stop at the first scope where we find the variable



Scope in Action

- This means that if there was an x before the global scope, we would have returned that value
- We stopped before our fourth scope, the built-in scope, which would recognize keywords such as True, list, ect.

Scope in Action

- This means that if there was an x before the global scope, we would have returned that value
- We stopped before our fourth scope, the built-in scope, which would recognize keywords such as True, list, ect.

```
x = 5

def enclosing():

    y = 6

    def local():

        z = x + y
```

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

Globals

X	6
Z	4

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```


Globals

X	6
Z	4

Locals

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```


Globals

X	6
Z	4

Locals

X	1
Y	

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

Globals

X	6
Z	4

Locals

X	1
Y	

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

Globals

X	6
Z	4

Locals

X	1
Y	

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

Globals

X	6
Z	4

Locals

X	1
Y	1+4

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

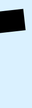
X	6
Z	4

Locals

X	1
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```



5

Globals

X	6
Z	4

Locals

X	1
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	6
Z	4

Locals

X	1
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	6
Z	4

Locals

X	2
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	6
Z	4

Locals

X	2
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	6
Z	4

Locals

X	2
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	6
Z	4

Enclosed

X	2
Y	5

Locals

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```


5

Globals

X	6
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	6
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```


5

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

Globals

X	8
Z	4

Enclosed

X	2
Y	5

Locals

Z	8

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

Globals

X	8
Z	4

Locals

X	2
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

Globals

X	8
Z	4

Locals

X	2
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

Globals

X	8
Z	4

Locals

X	10
Y	5

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

Globals

X	8
Z	4

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

Globals

X	8
Z	4

5

16

13

Scope Practice

```
x = 6
z = 4

def enclosing():
    x = 1
    y = x + z
    print(y)
    def localzz():
        z = 8
        global x
        x += 2
        print(z + x)
        print(y + x)

    x += 1
    localzz()
    x = 10

enclosing()
print(x)
```

5

16

13

8

Globals

X	8
Z	4

Pass by Reference? Pass by Value?

Immutable Objects

- The object itself CANNOT be modified
- Any change makes a new object

  - Int
  - Float
  - Long
  - Complex
  - Str
  - Tuple

Mutable Objects

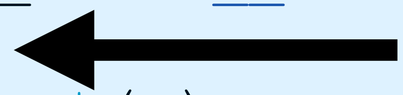
- The object itself can be modified
    - List
    - Set
    - Dict

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```

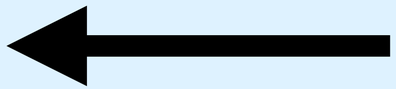


Globals

x	5

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```

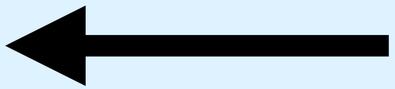


Globals

x	5

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```



Globals

x	5

Local

x	5

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```


Globals

x	5

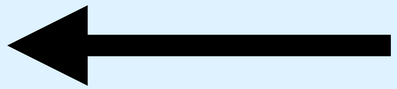
Local

x	6

This creates a whole new object, because ints are immutable, so this does not have any affect on the global x

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```



Globals

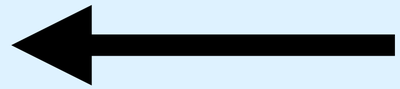
x	5

Local

x	6

Passing Immutable Objects

```
def increment(x):
    x = x + 1
    return

if __name__ == "__main__":
    x = 5
    increment(x)
    print(x)
```



Globals

x	5

5

Let’s check out what this looks like with mutable objects

Passing Mutable Objects

```
def append_one(arr):
    arr.append(1)
    return

if __name__ == "__main__":
    arr = [3, 2]
    append_one(arr)
    print(arr)
```


Globals

arr	[3,2]

Passing Mutable Objects

```
def append_one(arr):
    arr.append(1)
    return

if __name__ == "__main__":
    arr = [3, 2]
    append_one(arr)
    print(arr)
```


Globals

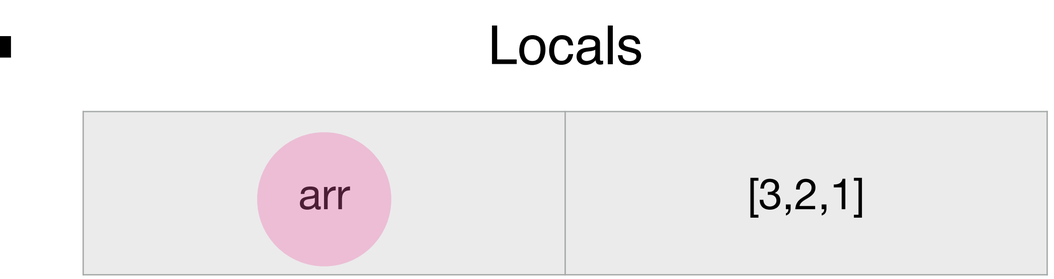
arr	[3,2]

Passing Mutable Objects

```
def append_one(arr):
    arr.append(1)
    return

if __name__ == "__main__":
    arr = [3, 2]
    append_one(arr)
    print(arr)
```

Passing Mutable Objects

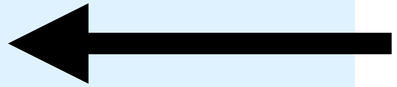
```
def append_one(arr):
    arr.append(1)
    return

if __name__ == "__main__":
    arr = [3, 2]
    append_one(arr)
    print(arr)
```



When you try to edit this variable, python does not need to create a new object, so the change permeates to the original one

Passing Mutable Objects

```
def append_one(arr):
    arr.append(1)
    return

if __name__ == "__main__":
    arr = [3, 2]
    append_one(arr)
    print(arr)
```



Globals

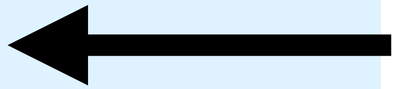
arr	[3,2,1]

Locals

arr	[3,2,1]

Passing Mutable Objects

```
def append_one(arr):
    arr.append(1)
    return

if __name__ == "__main__":
    arr = [3, 2]
    append_one(arr)
    print(arr)
```



Globals

arr	[3,2,1]

Locals

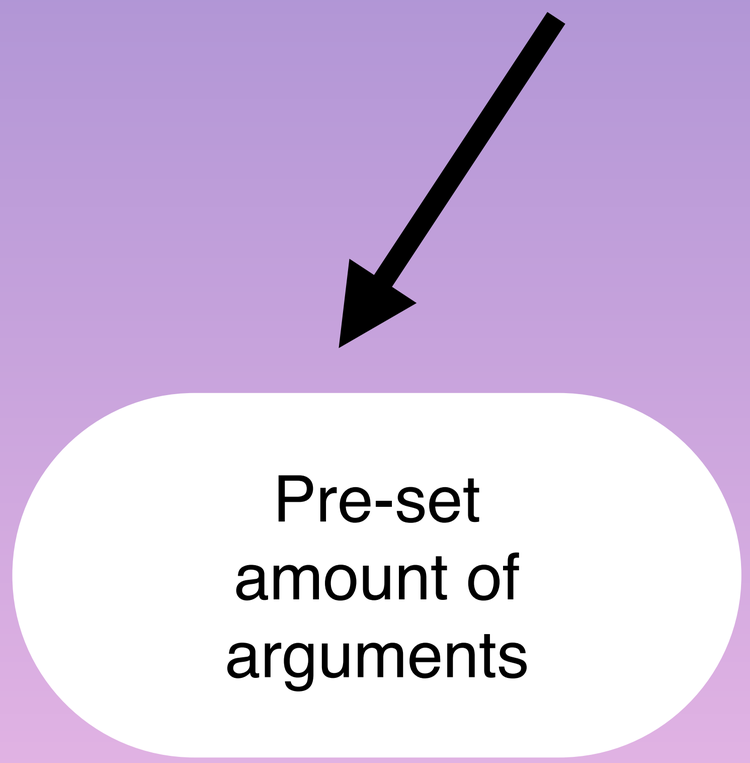
arr	[3,2,1]

3,2,1

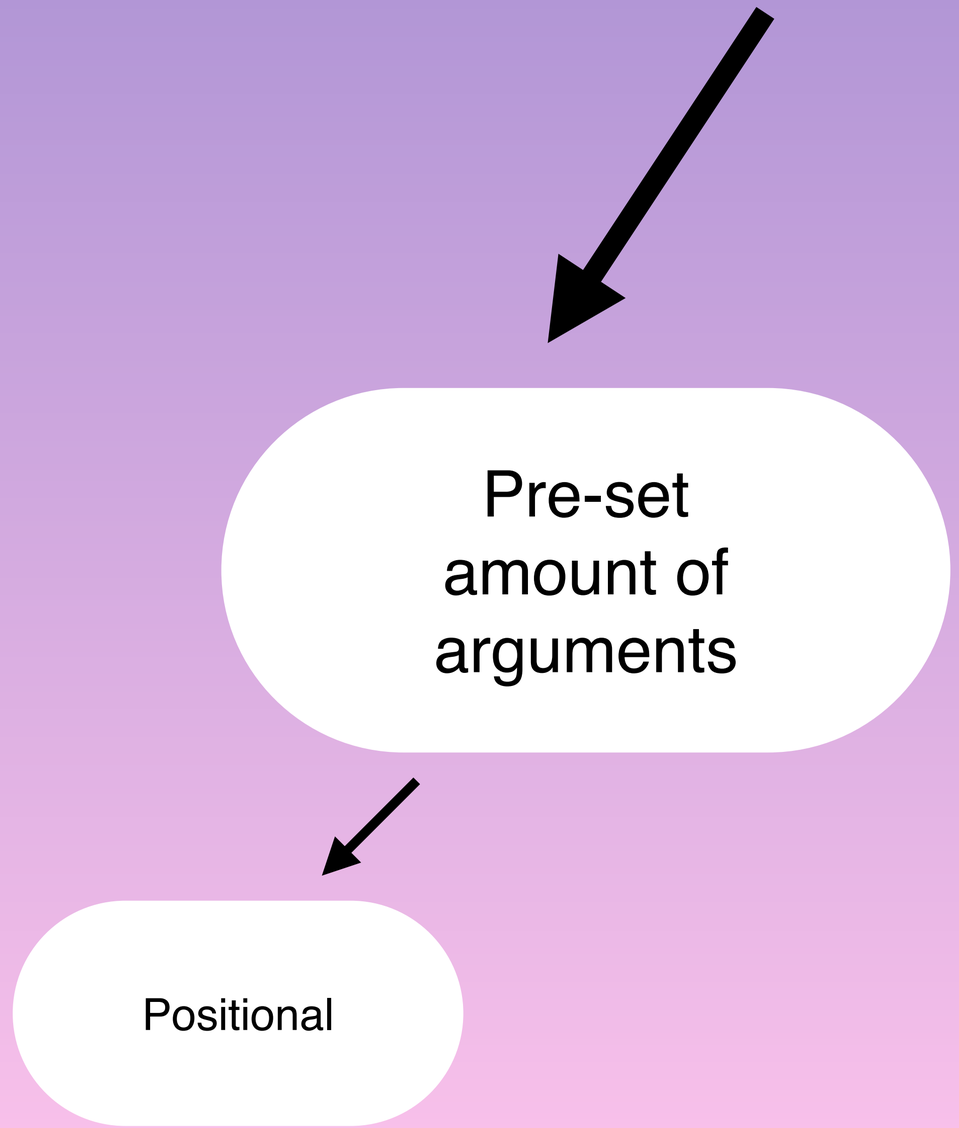
Function Arguments

a.k.a. parameters

Function Arguments



Function Arguments

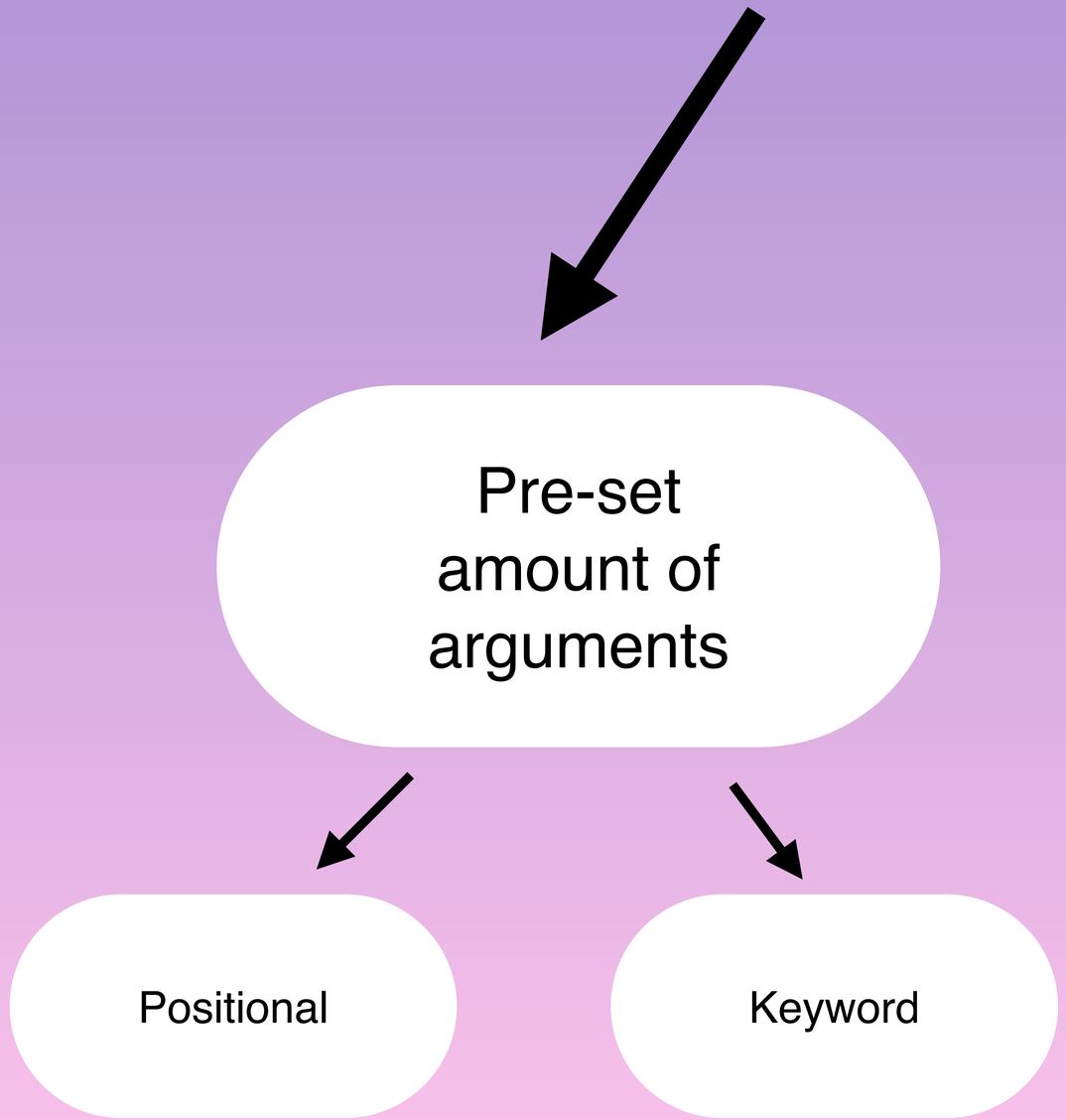


Positional

```
def euclidean_dist(x, y, z):
    return math.sqrt(x**2 + y**2 + x**2)

if __name__ == "__main__":
    euclidean_dist(5, 4, 3)
```

*W*hen calling a function, positional arguments are referenced by their *position in the function definition*, and are arguments of the following form. When calling a function, the caller is required to provide all positional arguments to the function.

Function Arguments



Keyword

```
euclidean_dist(y=4, x=5, z=3)
```

When calling a function, keyword arguments are referenced by argument name, and do not need to be provided in a specific order.

The below example presents a call to euclidean_dist in which we reference the arguments as keyword arguments, rather than positional arguments. You'll observe that we can call the arguments in any order.

Combining these two

```
euclidean_dist(3, 4, z=5)       # A valid call
euclidean_dist(x=5, 4, 3)       # An invalid call
euclidean_dist(x=5, y=4, z=3) # A valid call
```

When calling a function, Python enforces that positional arguments must appear in the function call before keyword arguments.

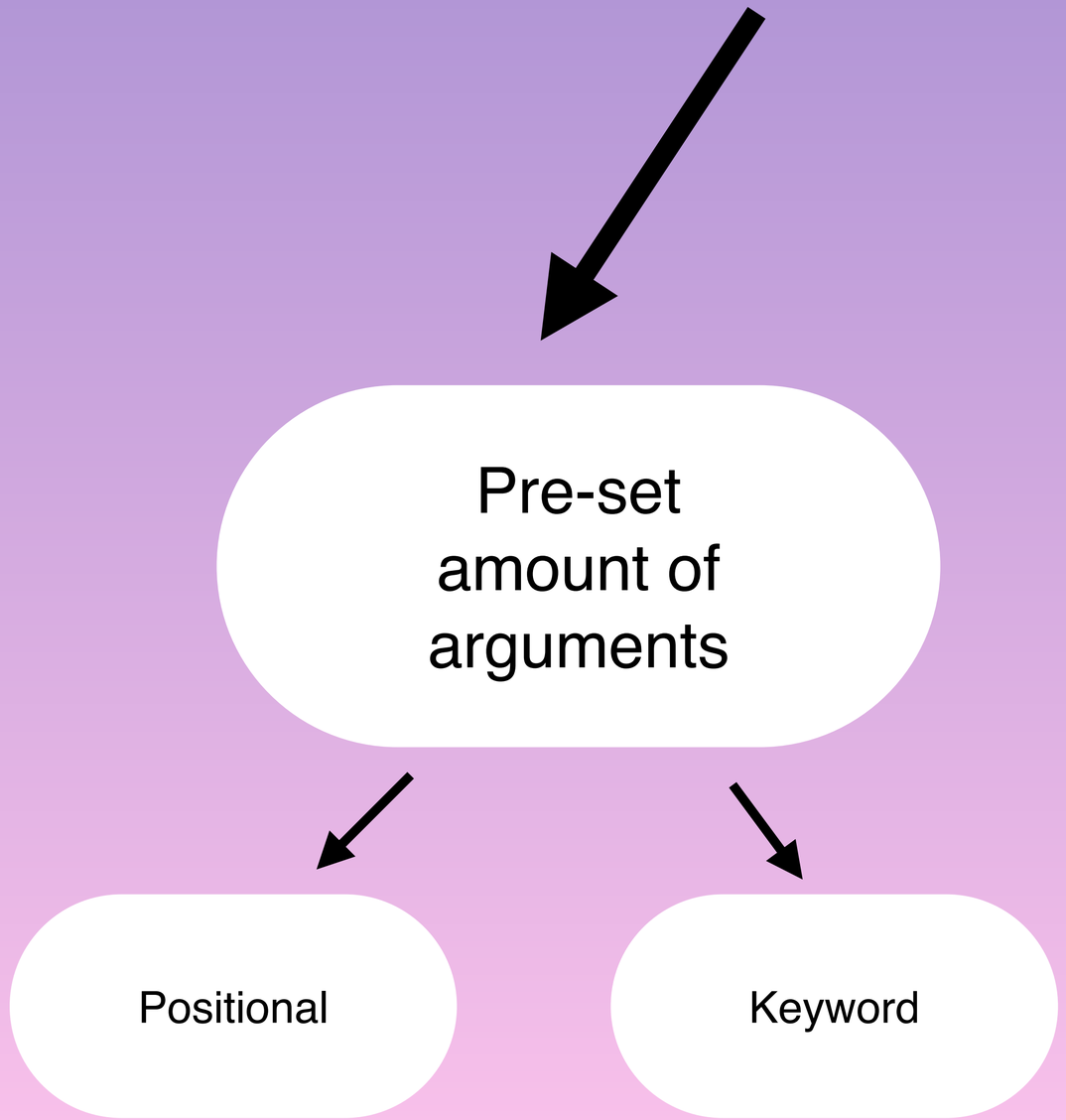
Positional Only

```
def euclidean_dist(x, y, /, z):
    return math.sqrt(x**2 + y**2 + x**2)

if __name__ == "__main__":
    euclidean_dist(5, 4, 3)         # A valid call
    euclidean_dist(5, 4, z=3)       # A valid call
    euclidean_dist(x=5, y=4, z=3) # Invalid call
```

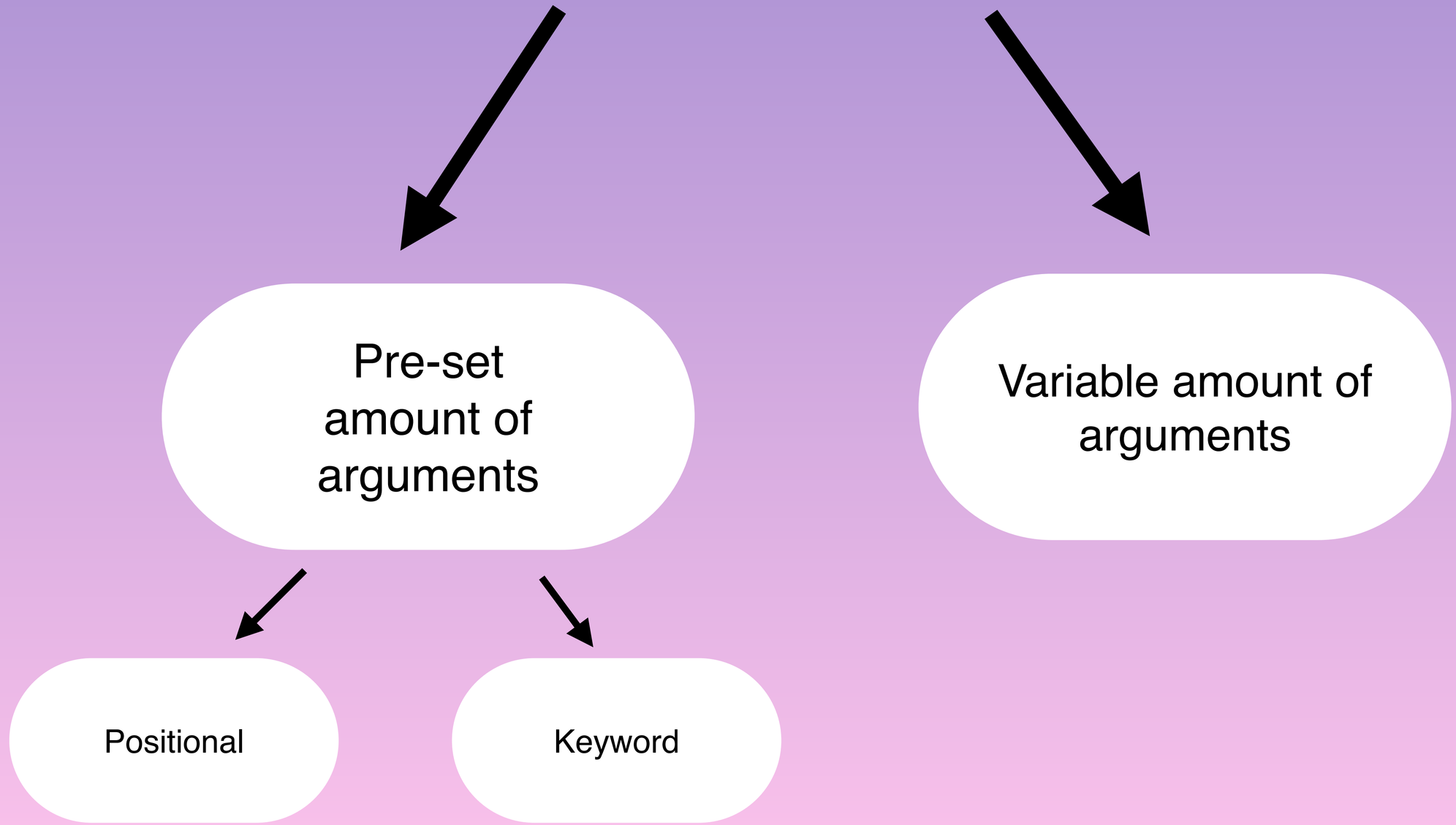
Keyword-Only

```
def euclidean_dist(x, y, *, z):
    return math.sqrt(x**2 + y**2 + x**2)

if __name__ == "__main__":
    euclidean_dist(5, 4, z=3)      # A valid call
    euclidean_dist(z=3, x=5, y=4) # A valid call
    euclidean_dist(5, 4, 3)        # Invalid call
```

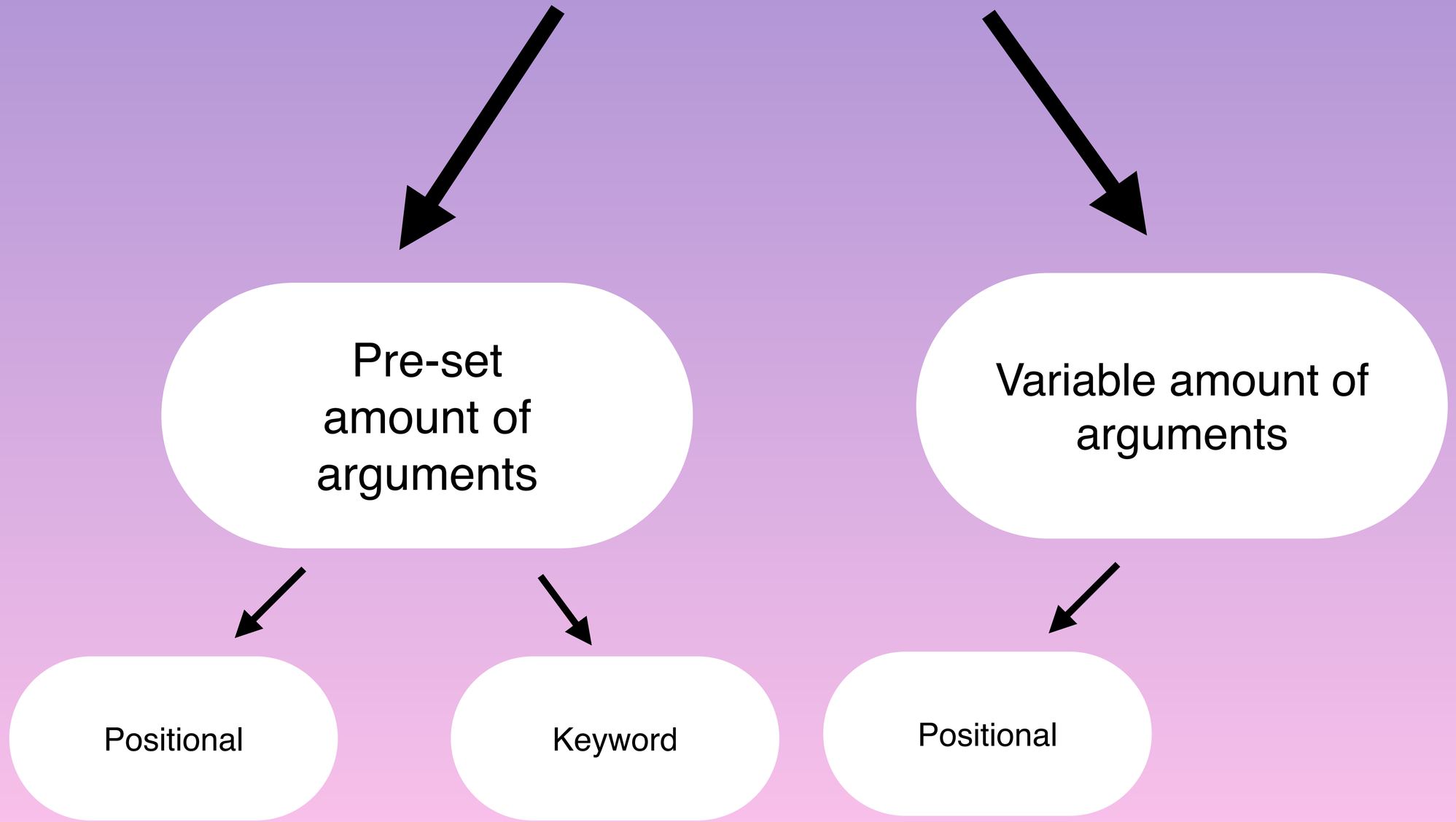
Function Arguments



Function Arguments



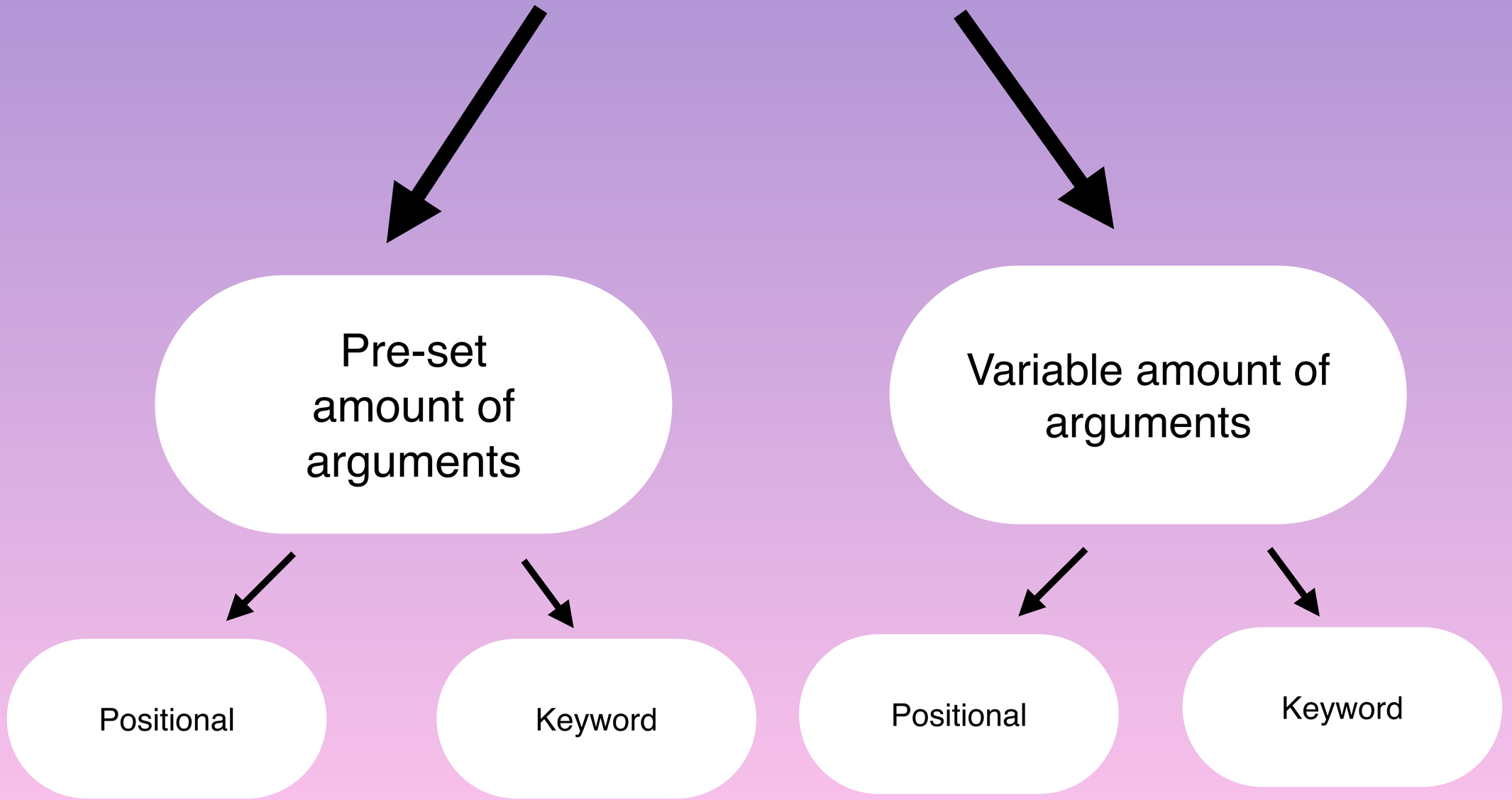
Function Arguments



Positional *Args

```
def greet_all(*names):
    for name in names:
        print("Hello {}!".format(name))

if __name__ == "__main__":
    greet_all("Parth", "Tara", "Theo")   # => Hello Parth!
                                         #    Hello Tara!
                                         #    Hello Theo!
```

Function Arguments



Positional **kwargs

```
def favourite_animals(**kwargs):
    for name, animal in kwargs.items():
        print("{}'s favourite animal is the {}.".format(name, animal))

if __name__ == "__main__":
    favourite_animals(Tara="horse", Parth="unicorn")

 # => Tara's favourite animal is the horse.

 #    Parth's favourite animal is the unicorn.
```

Unpacking!

```
def product_sum(a, b, c):
    return a*(b+c)

if __name__ == "__main__":
    tup = (3, 4, 5)
    product_sum(*tup) # Equivalent to product_sum(3, 4, 5)
```

Unpacking!

```
def favourite_animals(name, animal):
    print("{}'s favourite animal is the {}.".format(name, animal))

if __name__ == "__main__":
    animals_names = {"name":"Tara", "animal":"horse"}
    favourite_animals(**animals_names) # Equivalent to
favourite_animals(name="Tara", annimal="Horse")
```

Functions can take in any object as a parameter

…..and functions themselves are objects…

…..does this mean…

Functions can be parameters to functions?!?!?!??!

Yes! First Class Functions!

```
def edit_list(fn, lst):
    return [fn(x) for x in lst]

def fun_polynomial(x):
    return x**2 - 3*x + 12

edit_list(fun_polynomial, [1, 2, 3, 4]) # => [10, 10, 12,
16]
```

Sending a function in as a parameter with fn ^
We also can return a function…..

Or do both!

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

Functions can also return a function

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

We create a function that takes in a function as fn

Functions can also return a function

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

This function creates a new function that can take in any arguments (specified by args, kwags)

Functions can also return a function

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

This inner function first prints the arguments

Functions can also return a function

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

Then it calls the functions you send in as fn, with these arguments

Functions can also return a function

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

Lastly, we return this new function we created

Functions can also return a function

```
def foo(a, b, c=1):
    return (a + b) * c

foo = print_arguments(foo)
foo(2, 1, c=3)

# Arguments: (2, 3) {'c': 1}
# 9
```

@decorator Syntax

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime

@print_arguments
def foo(x,y):
    print(x + y)


foo(5,6)
```

This function, which takes in a function and edits it, is called a decorator