## Lambda

`(lambda arguments: expression)`

```
x = (lambda a, b, c : a + b)(1, 2, 3) # a → 1, b → 2, c → 3
print(x)
# Similar to writing
def f(a, b, c):
   return a + b
print(f(1, 2, 3))
```

If there are 3 arguments, and you only give it 2 it will give a error.

```
(lambda x: x+1)(2)
```

Can use them as an `anonymous function inside another function`.

```
def myfunc(n):
  return lambda a : a * n
```

Function definition that `takes one argument`, and that argument will be `multiplied` with an unknown number.

```
def myfunc(n):
    return lambda a: a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

## Nested Lambda

You can nest lambda in another lambda

`(lambda a : lambda b : a + b)`

1st: Takes one argument, a

2nd: Returns a function (which takes one argument, b; and returns the result a + b)

`(lambda a : lambda b : lambda c : a + b + c)`

1st: Takes one argument, a

2nd & 3rd: Returns a function (which takes one argument, b; and returns another function which takes one argument, c, and returns the result a + b + c).

```
x = (lambda a : lambda b : a + b)(1) # a → 1
print(x)
print(x(3))

print("########")

y = (lambda a : lambda b : a + b)(1)(2) # a → 1, b → 2
print(y)
```

Consecutive applications of function is treated as **left associative**.

```
(lambda x: x (lambda y: y))(lambda z : z)(1)
# (x (lambday: y) {{ x → lambda z : z }}) (1)
# ((lambdaz : z) (lambday: y)) (1)
# (z {{ z → lambda y : y }}) (1)
# (lambday : y) (1)
# y{{ y → 1 }}
# 1
```