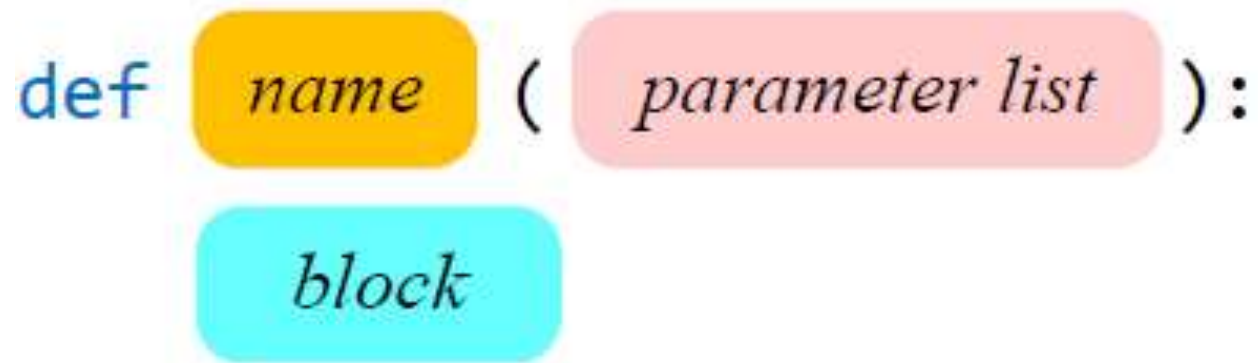


# Writing Functions

# Introduction

- There are two aspects to every Python function

**Function definition:** The definition of a function contains the code that determines the function's behavior.



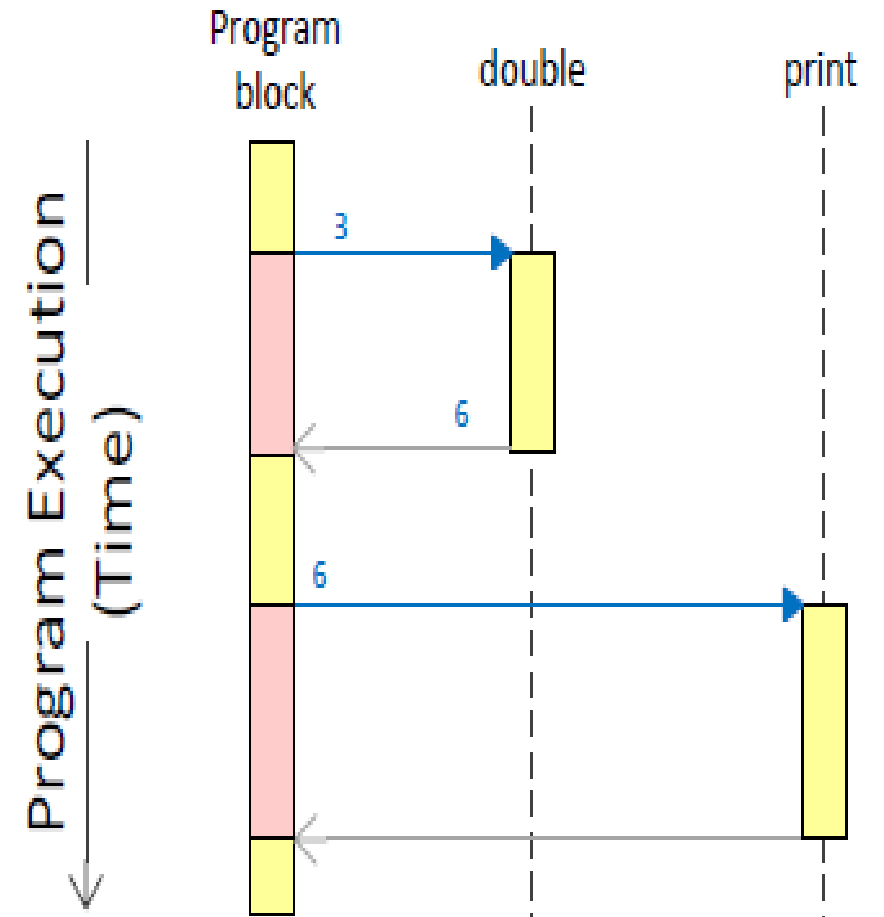
The diagram illustrates the syntax of a Python function definition. It shows the keyword `def` in blue, followed by a yellow box containing the word *name*, then an opening parenthesis `(`, a pink box containing the words *parameter list*, a closing parenthesis `)`, and a colon `:`. Below this line, a cyan box containing the word *block* is shown, representing the code block that follows the function definition.

```
def name ( parameter list ):  
    block
```

**Function invocation :** A function is used within a program via a function invocation.

# Example

```
def double(n):  
    return 2 * n # Return twice the given number  
  
# Call the function with the value 3 and print its result  
x = double(3)  
print(x)
```



# Example

```
# Compute the greatest common factor of two integers
# provided by the user

def gcd(n1, n2):
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i    # Found larger factor
    return largest_factor

# Exercise the gcd function

# Prompt user for input
num1 = int(input('Please enter an integer: '))
num2 = int(input('Please enter another integer: '))

# Print the GCD
print(gcd(num1, num2))
```

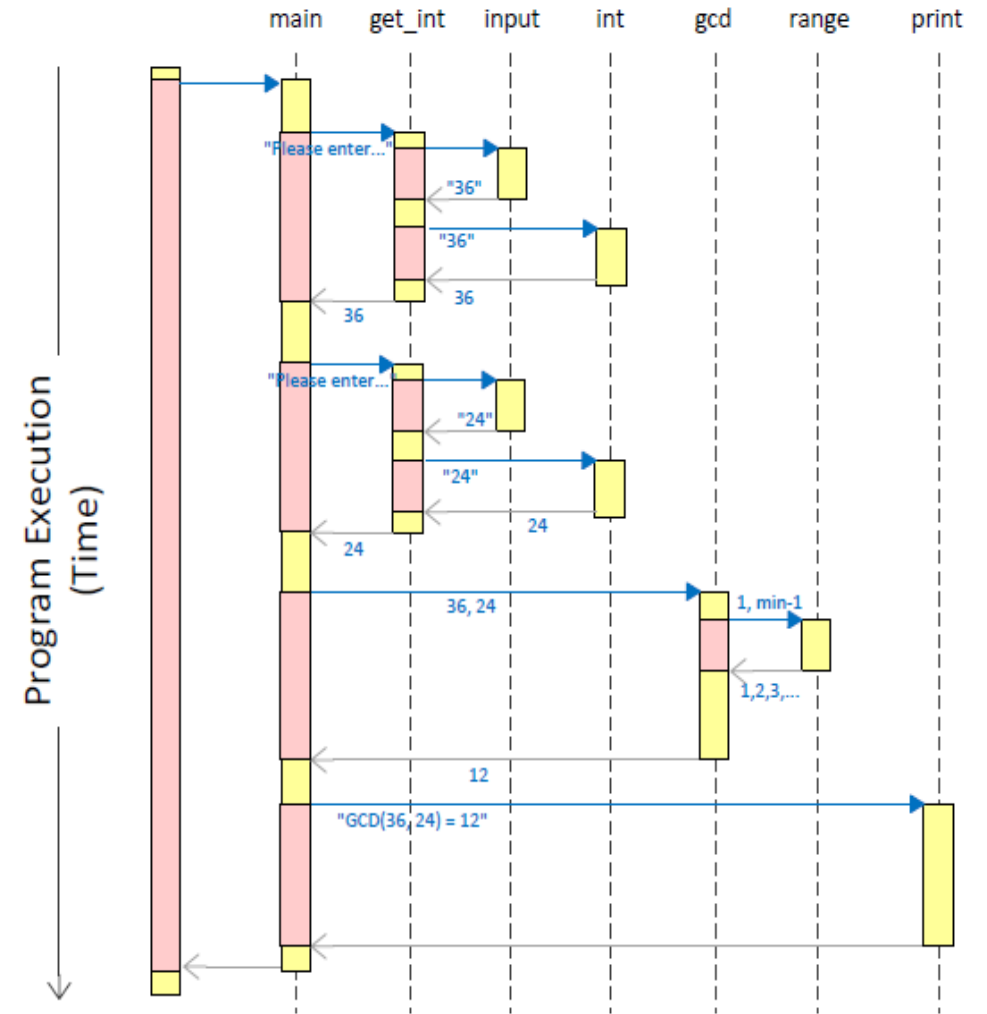
# Example

```
# Computes the greatest common divisor of m and n
def gcd(m, n):
    # Determine the smaller of m and n
    min = m if m < n else n
    # 1 is definitely a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if m % i == 0 and n % i == 0:
            largest_factor = i    # Found larger factor
    return largest_factor

# Get an integer from the user
def get_int():
    return int(input("Please enter an integer: "))

# Main code to execute
def main():
    n1 = get_int()
    n2 = get_int()
    print("gcd(", n1, ",", n2, ") = ", gcd(n1, n2), sep="")

# Run the program
main()
```



```
def help_screen():
    """
    Displays information about how the program works.
    Accepts no parameters.
    Returns nothing.
    """

    print("Add: Adds two numbers")
    print("Subtract: Subtracts two numbers")
    print("Print: Displays the result of the latest operation")
    print("Help: Displays this help screen")
    print("Quit: Exits the program")
```

```
def menu():
    """
    Displays a menu
    Accepts no parameters
    Returns the string entered by the user.
    """

    return input("== A)dd S)ubtract P)rint H)elp Q)uit ==")
```

```
def main():
    """ Runs a command loop that allows users to perform simple arithmetic. """
    result = 0.0
    done = False # Initially not done
    while not done:
        choice = menu() # Get user's choice

        if choice == "A" or choice == "a": # Addition
            arg1 = float(input("Enter arg 1: "))
            arg2 = float(input("Enter arg 2: "))
            result = arg1 + arg2
            print(result)
        elif choice == "S" or choice == "s": # Subtraction
            arg1 = float(input("Enter arg 1: "))
            arg2 = float(input("Enter arg 2: "))
            result = arg1 - arg2
            print(result)
        elif choice == "P" or choice == "p": # Print
            print(result)
        elif choice == "H" or choice == "h": # Help
            help_screen()
        elif choice == "Q" or choice == "q": # Quit
            done = True
```

main()

# Local Variables

- Variables defined within functions are local variables.
- Variable exists only during the function's execution.
- The memory required to store a local variable is used only when the variable is in scope.
- The same variable name can be used in different functions without any conflict

# Global Variables

- Variable created outside any function.
- The variable is declared to be a global variable using the `global` reserved word



# Default Parameters

We have seen how callers may invoke some Python functions with differing numbers of parameters. Compare

```
a = input()
```

to

```
a = input("Enter your name: ")
```

We can define our own functions that accept a varying number of parameters by using a technique known as default parameters.

```
def countdown(n=10):  
    for count in range(n, -1, -1):  
        print(count)
```

# Default Parameters

```
def sum_range(n, m=100):    # OK, default follows non-default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

and

```
def sum_range(n=0, m=100):    # OK, both default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

but the following definition is illegal, since a default parameter precedes a non-default parameter in the function's parameter list:

```
def sum_range(n=0, m):    # Illegal, non-default follows default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

# Introduction to Recursion

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$
$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot (n-1)!, & \text{otherwise.} \end{cases}$$

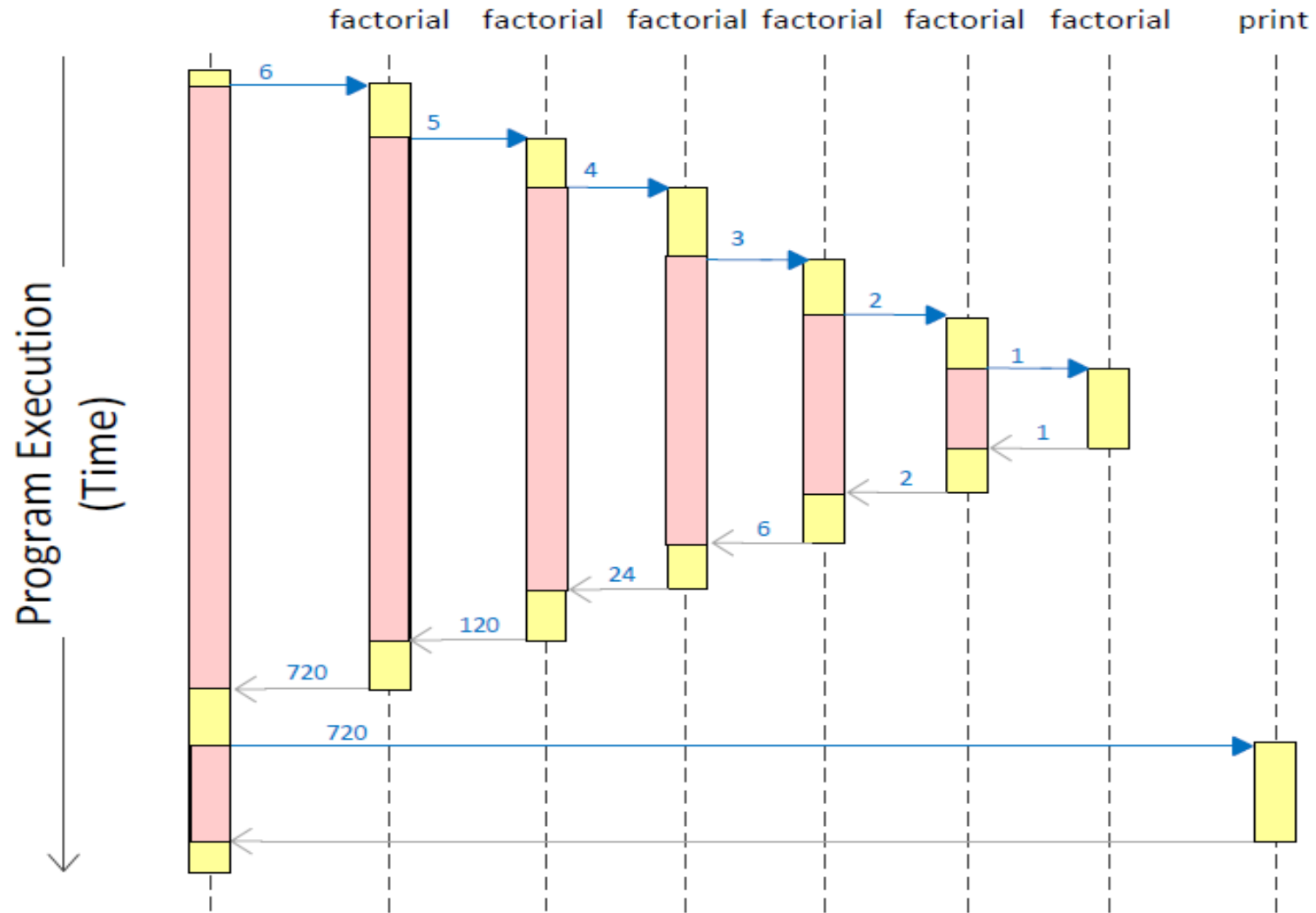
---

```
def factorial(n):  
    """  
    Computes n!  
    Returns the factorial of n.  
    """  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

# factorial

```
factorial(6) = 6 * factorial(5)
              = 6 * 5 * factorial(4)
              = 6 * 5 * 4 * factorial(3)
              = 6 * 5 * 4 * 3 * factorial(2)
              = 6 * 5 * 4 * 3 * 2 * factorial(1)
              = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
              = 6 * 5 * 4 * 3 * 2 * 1 * 1
              = 6 * 5 * 4 * 3 * 2 * 1
              = 6 * 5 * 4 * 3 * 2
              = 6 * 5 * 4 * 6
              = 6 * 5 * 24
              = 6 * 120
              = 720
```

# factorial



# Greatest Common Divisor

```
def gcd(m, n):  
    """  
    Uses Euclid's method to compute the greatest common divisor  
    (also called greatest common factor) of m and n.  
    Returns the GCD of m and n.  
    """  
    if n == 0:  
        return m  
    else:  
        return gcd(n, m % n)  
  
def iterative_gcd(num1, num2):  
    """  
    Uses a naive algorithm to compute the greatest common divisor  
    (also called greatest common factor) of m and n.  
    Returns the GCD of m and n.  
    """  
    # Determine the smaller of num1 and num2  
    min = num1 if num1 < num2 else num2  
    # 1 is definitely a common factor to all integers  
    largest_factor = 1;  
    for i in range(1, min + 1):  
        if num1 % i == 0 and num2 % i == 0:  
            largest_factor = i # Found larger factor  
    return largest_factor
```

```
def main():  
    """ Try out the gcd function """  
    for num1 in range(1, 101):  
        for num2 in range(1, 101):  
            print("gcd of", num1, "and", num2, "is", gcd(num1, num2))
```

main()

---

# Functions as Data

- In Python, a function is a special kind of **object**, just as integers and strings are objects.

```
>>> type(2)
<class 'int'>

>>> type('Rick')
<class 'str'>

>>> from math import sqrt
>>> type(sqrt)
<class 'builtin_function_or_method'>
```

# Functions as Data

- We also can pass a function as a parameter to another function.

```
def add(x, y):  
    """  
    Adds the parameters x and y and returns the result  
    """  
    return x + y  
  
def multiply(x, y):  
    """  
    Multiplies the parameters x and y and returns the result  
    """  
    return x * y  
  
def evaluate(f, x, y):  
    """  
    Calls the function f with parameters x and y:  
    f(x, y)  
    """  
    return f(x, y)
```

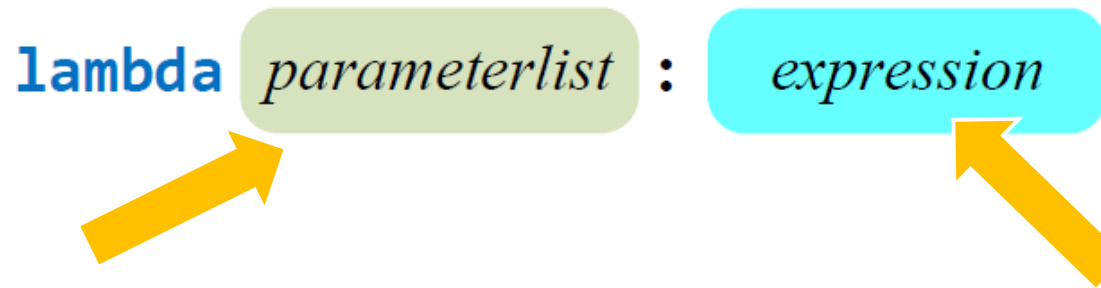


# Functions as Data

```
def main():  
    """  
    Tests the add, multiply, and evaluate functions  
    """  
    print(add(2, 3))  
    print(multiply(2, 3))  
    print(evaluate(add, 2, 3))  
    print(evaluate(multiply, 2, 3))  
  
main() # Call main
```

# Lambda Expressions

- What if we want to ensure that our function will execute exactly one time and only when invoked by another function?
- Python supports the definition of simple, **anonymous functions** via **lambda** expressions.



A comma-separated  
list of parameters

A single Python expression

We can use a **lambda** expression in a call to the `evaluate` function:

```
evaluate(lambda x, y: x * y, 2, 3)
```

# Lambda Expressions

```
1  def cube(y):  
2      return y*y*y  
3  
4  
5  lambda_cube = lambda y: y*y*y  
6  
7  
8  # using function defined  
9  # using def keyword  
10 print("Using function defined with `def` keyword, cube:", cube(5))  
11  
12 # using the lambda function  
13 print("Using lambda function, cube:", lambda_cube(5))
```

# Lambda Expressions

With lambda function	Without lambda function
Supports single line statements that returns some value.	Supports any number of lines inside a function block
Good for performing short operations/data manipulations.	Good for any cases that require multiple lines of code.
Using lambda function can sometime reduce the readability of code.	We can use comments and function descriptions for easy readability.

# Generators

- A generator is a programming object that produces (that is, generates) a **sequence of values**.
- Code that uses a generator may obtain **one value in the sequence** at a time.

```
def gen():  
    yield 3  
    yield 'wow'  
    yield -1  
    yield 1.2
```

```
for i in gen():  
    print(i)
```

```
3  
wow  
-1  
1.2
```

# Generators

```
1  def generate_multiples(m, n):
2      count = 0
3      while count < n:
4          yield m * count
5          count += 1
6
7  def main():
8      for mult in generate_multiples(3, 6):
9          print(mult, end=' ')
10     print()
11
12  main()
```

# Generators

```
1  def myrange(arg1, arg2=None, step=1):
2      if arg2 != None:
3          begin = arg1
4          end = arg2
5      else:
6          begin = 0
7          end = arg1
8
9      i = begin
10     while i != end:
11         yield i
12         i += step
```