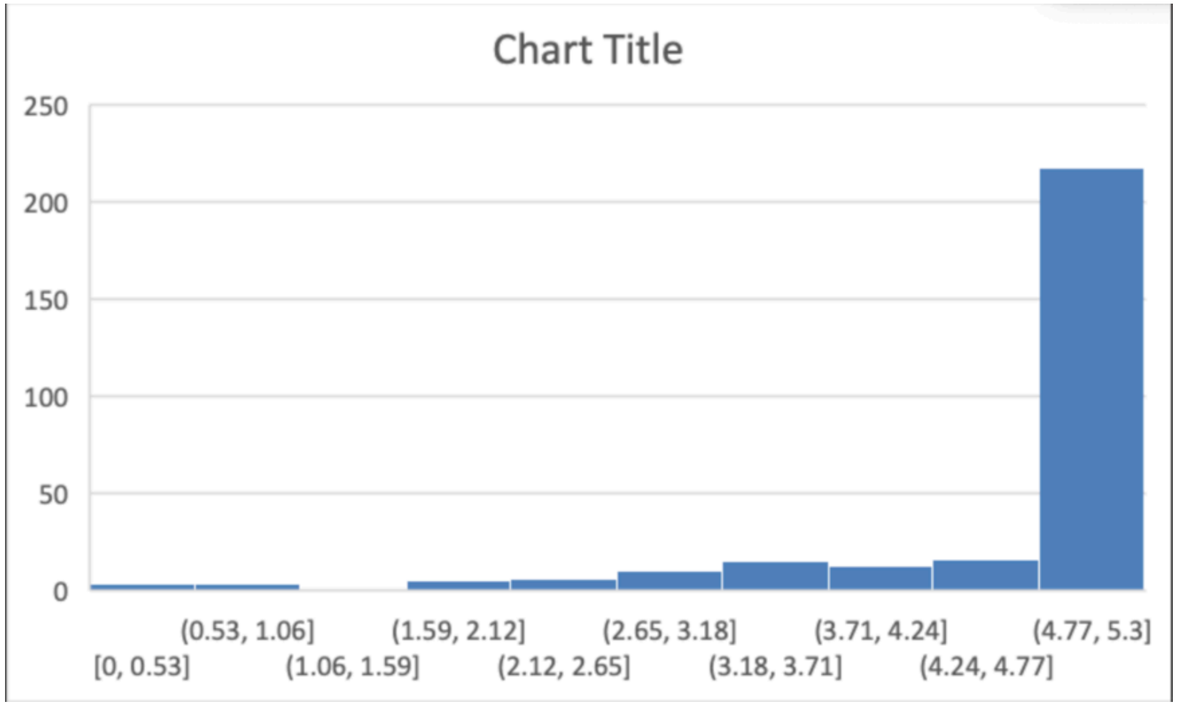
COL1000: Introduction to Programming

Functions

Subodh Sharma | Lec 17 | Sept 19

Reminders!

- **Refer to the other instructors' slides and lecture code for more practice and ideas!**
- **Lab Test 1 performance**
  - Avg: 4.55
  - Median: 5



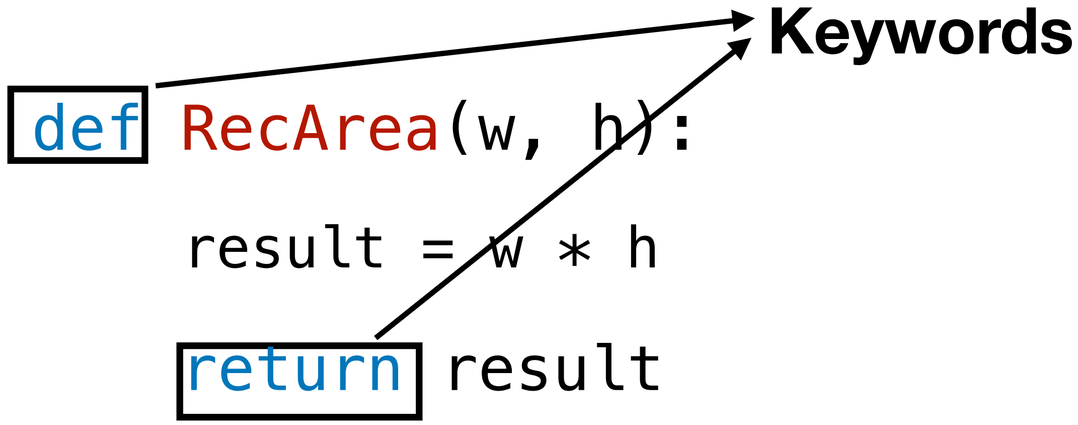
Functions: The Art of Programming

Functions

What and Why?

- Functions are **named regions of code**
  - **Eg:** print, input, input.split(), str.strip() … etc.
- Functions are **tools for abstractions**
  - Take a bunch of execution steps (achieving some functionality) and put them in the code as a single “abstract” executable step
    - Eg: Drawing a square of length 1 unit from A: **Draw_Perpendicular(A)**, ….. , **Draw_Perpendicular(B)**, …
- **Why — Reuse, Reduce code, Readability & Maintainability, Units for testing**

Functions — Syntax (Definition)



- **def**: keyword that introduces the function code
- **return:** sends a value back (default is `None`, if no return statement) (**functions are first-class objects and as such treated as vars**)
- In addition, **name of the function** has to be provided. Eg: `RecArea`
- (Optional) **Parameters:** Placeholder variables that the functions uses for computation. Eg: `w,h`
- **Body of the function:** The code within the **named scope!**

Functions — Syntax (Invocation)

Parameters vs (Positional) Arguments

```
def RecArea(w, h):
    result = w * h
    return result
```

```
if __name__ == “__main__”:
    res = RecArea(3,4)
    print(“res”)
```

- **Function invocation:** `RecArea` is called with concrete **arguments**
  - These are **positional arguments; Changing the order will change the assignment!**
- Each module has a builtin var: **`__name__`**. If you run a python file, then for that file `__name__ == __main__` is `True`
-

Functions — Syntax (Definition & Invocation)

Default Parameters

```
def RecArea(w, h = 1):
    result = w * h
    return result
```

```
if __name__ == “__main__”:
    res = RecArea(3)
    print(“res”)
```

- **Altenative definition:** with default values of parameters
  - Non-default params must come before default params — **otherwise a syntax error**!
  - Be **cautious** while supplying **defaults as mutable** entities!
    - It may lead to errors!

Functions — Syntax (Definition & Invocation)

Positional Arguments vs Keyword Arguments

```
def RecArea(w, h = 1):
    result = w * h
    return result
```

```
if __name__ == “__main__”:
    res = RecArea(h=3, w=1)
    print(“res”)
```

- **Altenative definition:** overriding default values of parameters, but with **keyword arguments**
  - Order of parameters not imporant
  - Keyword arguments must always follow all positional arguments — **else syntax error**
- **Example of Positional Args and KWArgs**
  - Note what data structures are used for Args and KWArgs

Functions — Semantics

Scope - Local, Global, Nonlocal

- **Local scope:** Vars defined inside a function are accessible within that function

```
def local_scope_example():
    x = 10  # local variable
    print(x)
```

- **Enclosing scope:** In nested functions, inner functions can access vars from the outer enclosing function

```
def outer():
    x = 'outer variable'

    def inner():
        print(x)  # accessing enclosing variable

    inner()
```

- **Global Scope:** Vars defined outside all functions have global scope

Functions — Semantics

Scope - Local, Global, Nonlocal

```
# total = 0
def make_adder(k): # example of higher order function
    #global total
    total = 0
    def add(x):
        nonlocal total    # modify enclosing scope
        total += x
        return x + k
    return add

print(make_adder(5)(3))
```

- Try declaring total as a global and modifying it in the nested functions

Functions — Semantics

Scope - Local, Global, Nonlocal

- **Local scope:** Vars defined inside a function are accessible within that function

```
def local_scope_example():
    x = 10  # local variable
    print(x)
```

- **Enclosing scope:** In nested functions, inner functions can access vars from the outer enclosing function

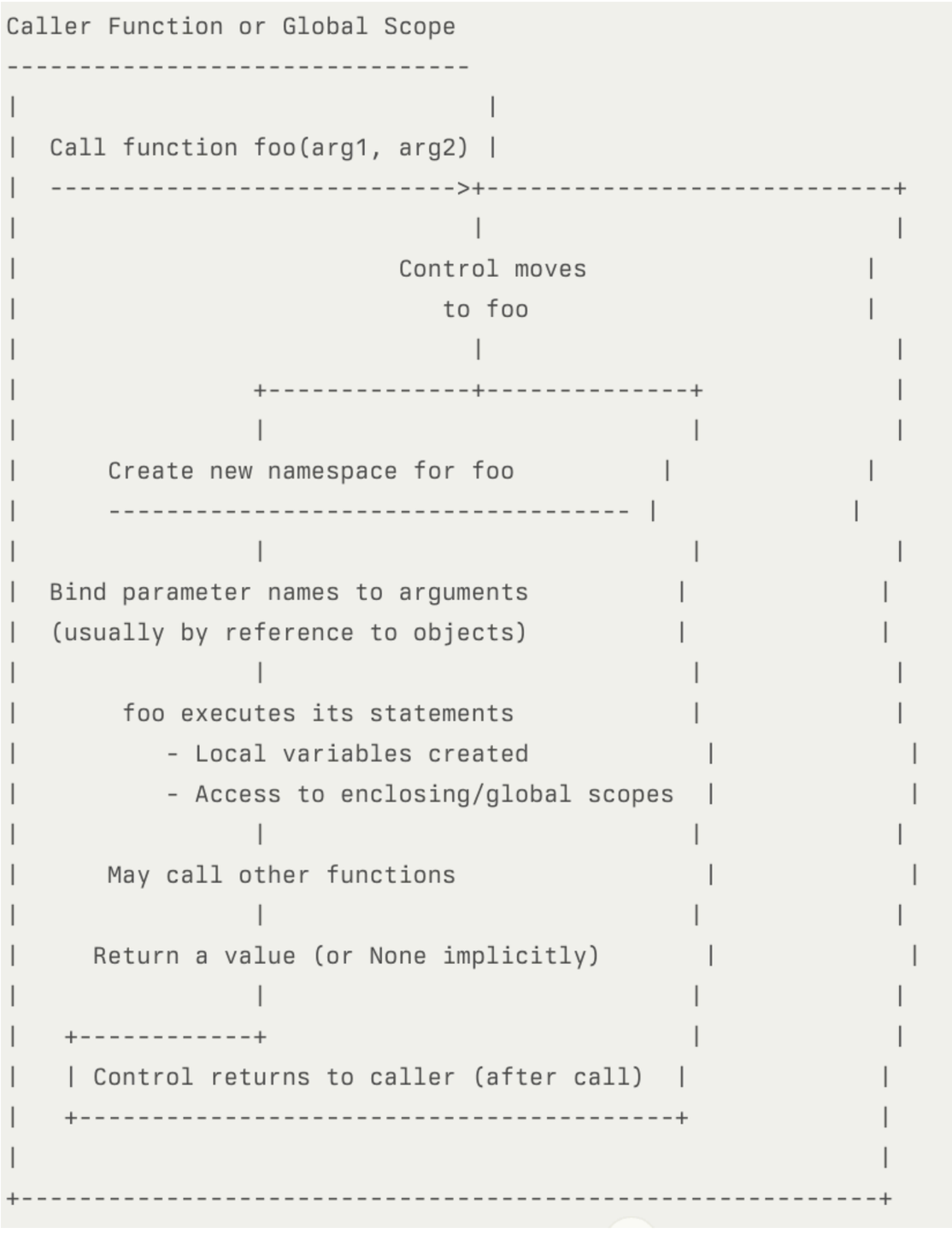
```
def outer():
    x = 'outer variable'

    def inner():
        print(x)  # accessing enclosing variable

    inner()
```

- **Global Scope:** Vars defined outside all functions have global scope

Functions — Semantics

Execution Semantics

- **Namespace for the callee** — creation of local scope
  - Each new function invocation creates a frame on the memory stack
- Parameter binding with arguments
  - **Either by reference or by value**



Functions — Semantics

Parameter Binding — Pass by Object Reference

- **In Python —** when an var is passed to a function as an argument, its reference is created (referring to the same object) and transferred to the callee's namespace!
  - In that sense it is neither transfer of value or direct reference.
- **Case 1: When the passed object is immutable**
  - Since the reference cannot be changed, any modification creates a copy of the object and the passed reference now points to the modified object
- **Case 2: When the passed object is mutable**
  - The passed reference points to the same reference with which the function was invoked

Functions as First-class objects

What does it mean?

- First-class => Functions can be
  - **Stored in to variables** or data structures like lists, etc.
  - **Passed as arguments** to other functions (Eg: filters, accumulators etc.)
  - Can be **returned** from another functions

Functions: Closures

- A **closure** is a function object that **remembers values from its enclosing scope**

```
def make_gpa():
    total_points = 0.0
    total_credits = 0.0
    def add_course(grade_point, credits):
        nonlocal total_points, total_credits
        total_points  += grade_point * credits
        total_credits += credits
        return total_points / total_credits
    return add_course

gpa = make_gpa()
print(gpa(8.0, 4))    # 8.0
print(gpa(9.0, 3))    # 8.428. Also an example of closure
```