Function (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Function (1)

We use functions to break up our code into small chunks.
These chunks are easier to read, understand and maintain.
If there are bugs, it's easier to find bugs
in a small chunk than the entire program.
We can also re-use these chunks.

```
def greet_user(name):
    print(f"Hi {name}")

greet_user("John")
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Formatted string literals

The **f** means Formatted string literals and it's new in Python 3.6.

A formatted string literal or f-string is
a string literal that is prefixed with **f** or **F**.

These strings may contain replacement fields,
which are expressions delimited by curly braces **{}**.
While other string literals always have a constant value,
formatted strings are really expressions evaluated at run time.

```
    print(f"Hi {name}")
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Argument types

Parameters are placeholders for the data
we can pass to functions.

Arguments are the actual values we pass.

We have two types of arguments:

- **positional arguments**:
  their position (order) matters

- **keyword arguments**:
  position doesn't matter
  prefix them with the parameter name.

```
# Two positional arguments
greet_user("John", "Smith")

# Keyword arguments
calculate_total(order=50, shipping=5, tax=0.1)
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Function (3)

Our functions can return values.

If we don't use the return statement,
by default, **None** is returned.

None is an object that represents
the absence of a value.

```
def square(number):
    return number * number

result = square(2)
print(result)
```

https://programmingwithmosh.com/wp-content/uploads/2019/02/Python-Cheat-Sheet.pdf

Function Arguments

A function can take arguments and return values:

the **function** **say_hello**
receives the **argument** "name" and
prints a greeting:

```
>>> def say_hello(name):
...     print(f'Hello {name}')
...
>>> say_hello('Carlos')
# Hello Carlos

>>> say_hello('Wanda')
# Hello Wanda

>>> say_hello('Rose')
# Hello Rose
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Keyword Arguments

To improve code readability,
we should be as explicit as possible.

by using Keyword Arguments:

```
>>> def say_hi(name, greeting):
...     print(f"{greeting} {name}")
...
>>> # without keyword arguments
>>> say_hi('John', 'Hello')
# Hello John

>>> # with keyword arguments
>>> say_hi(name='Anna', greeting='Hi')
# Hi Anna
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Return Values

A return statement consists of

The **return** keyword.

The **value** or expression that the function should return.

```
>>> def sum_two_numbers(number_1, number_2):
...     return number_1 + number_2
...
>>> result = sum_two_numbers(7, 8)
>>> print(result)
# 15
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Local and Global Scope

Code in the global scope
cannot use any local variables.

a local scope
can access global variables.

Code in a function's local scope
cannot use variables in any other local scope.

You can use the same name for different variables
if they are in different scopes.

That is, there can be a local variable named spam
and a global variable also named spam.

https://www.pythoncheatsheet.org/cheatsheet/functions

Function (8)

```
global_variable = 'I am available everywhere'

>>> def some_function():
...     print(global_variable)  # because is global
...     local_variable = "only available within this function"
...     print(local_variable)
...
>>> # the following code will throw error because
>>> # 'local_variable' only exists inside 'some_function'
>>> print(local_variable)
Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
NameError: name 'local_variable' is not defined
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Global statement

If you need to modify a global variable
from within a function,
use the global statement:

```
>>> def spam():
...     global eggs
...     eggs = 'spam'
...
>>> eggs = 'global'
>>> spam()
>>> print(eggs)
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Scope rules

If a variable is being used in the global scope
(that is, outside all functions),
then it is always a global variable.

If there is a global statement
for that variable in a function,
it is a global variable.

Otherwise, if the variable is used
in an assignment statement in the function,
it is a local variable.

But if the variable is not used
in an assignment statement,
it is a global variable.

https://www.pythoncheatsheet.org/cheatsheet/functions

Lambda functions

In Python, a lambda function is
a single-line, anonymous function,
which can have any number of arguments,
but it can only have one expression.

From the Python 3 Tutorial

lambda is a minimal function definition
that can be used inside an expression.

Unlike FunctionDef, body holds a single node.

Single line expression

Lambda functions can only
evaluate an expression,
like a single line of code.

https://www.pythoncheatsheet.org/cheatsheet/functions

Lambda function examples

```
>>> def add(x, y):
...     return x + y
...
>>> add(5, 3)
# 8
```

the equivalent lambda function:

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
# 8
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Lambda functions as lexical closures

Like regular nested functions, lambdas also work as lexical closures:

```
>>> def make_adder(n):
...     return lambda x: x + n
...
>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
# 7
>>> plus_5(4)
# 9
```

https://www.pythoncheatsheet.org/cheatsheet/functions

Arbitaray Arguments *args

If you do not know how many arguments
that will be passed into your function,
add a * before the parameter name
in the function definition.

This way the function will
receive a tuple of arguments, and
can access the items accordingly:

If the number of arguments is unknown,
add a * before the parameter name:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

https://www.w3schools.com/python/python_functions.asp

Arbitrary Keyword Arguments **kwargs

If you do not know how many keyword arguments
that will be passed into your function,
add two asterisk: ** before the parameter name
in the function definition.

This way the function will
receive a dictionary of arguments, and
can access the items accordingly:

If the number of keyword arguments is unknown,
add a double ** before the parameter name:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

https://www.w3schools.com/python/python_functions.asp

Default Parameter Value

If we call the function without argument,
it uses the default value:

```
def my_function(country = "Norway"):
    print("I am from " + country)
```

```
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
my_function("Norway")
```

https://www.w3schools.com/python/python_functions.asp

Passing a List as an Argument

You can send any data types of argument
to a function (string, number, list, dictionary etc.), and
it will be treated as the same data type inside the function.

if you send a List as an argument,
it will still be a List when it reaches the function:

```
def my_function(food):
    for x in food:
        print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

https://www.w3schools.com/python/python_functions.asp

The pass Statement

function definitions cannot be empty,
but if you for some reason have a function definition with no content,
put in the **pass** statement to avoid getting an error.

```
def myfunction():
    pass
```

https://www.w3schools.com/python/python_functions.asp

Recursion (1)

Recursion is a common mathematical and programming concept.
It means that a function calls itself.
This has the benefit of meaning
that you can loop through data to reach a result.

very careful with recursion as it can be quite easy
to slip into writing a function which never terminates,
or one that uses excess amounts of memory or processor power.

However, when written correctly recursion
can be a very efficient and
mathematically-elegant approach to programming.

https://www.w3schools.com/python/python_functions.asp

Recursion (2)

In this example, **tri_recursion()** is a function
that we have defined to call itself ("recurse").

We use the **k** variable as the data,
which decrements (-1) every time we recurse.

The recursion ends
when the condition is not greater than 0 (i.e. when it is 0).

```
def tri_recursion(k):
    if (k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

https://www.w3schools.com/python/python_functions.asp

Recursion (3)

```
tri_recursion(6)
  6 + tri_recursion(5)
        5 + tri_recursion(4)
              4 + tri_recursion(3)
                    3 + tri_recursion(2)
                          2 + tri_recursion(1)
                                1 + tri_recursion(0)
                                      return 0
                                return 1
                          return 3
                    return 6
              return 10
        return 15
return 21
```

https://www.w3schools.com/python/python_functions.asp

Lexical Closures

```
def makeAdder(addend):
        def adder(augend):
                return augend + addend
        return adder

add23 = makeAdder(23)
add42 = makeAdder(42)

print add23(100),add42(100),add23(add42(100))
123 142 165
```

https://www.w3schools.com/python/python_functions.asp

References

[1] Essential C, Nick Parlante
[2] Efficient C Programming, Mark A. Weiss
[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
[4] C Language Express, I. K. Chun