Functions Cheat Sheet

Python Functions

Introduction to Functions

```
# Use def to define new functions
def my_function1():
    """
     This is the function's docstring.
    """
    print('This is the function's body!')
    # this function returns None implicitly

# Calling the function
my_function1()            # => This is a function!
my_function1.__doc__      # => This is the function's docstring.

# The return statement exits the function
def my_function2():
    x = 1
    return x                            # the function ends here
    print('Never reaches this line!')   # it will never reach this line

# Calling the function
my_function2()      # returns 1

# A function can return more values as a tuple
def add_multiply_power(x, y):
    return x + y, x * y, x ** y

# Calling the function
a, b, c = add_multiply_power(2, 3)    # returns (2 + 3, 2 * 3, 2 ** 3)
print(a, b, c)                         # => 5 6 8
```

Function's Arguments

```
# 1. Function with positional arguments
def add(x, y):
   print(f"x is {x} and y is {y}")
   return x + y    # returns the result of x + y

# Calling function with positional arguments
s = add(5, 6)    # => prints out "x is 5 and y is 6" and returns 11, s is 11

# Calling function with keyword arguments
s = add(y=1, x=8)    # => prints out "x is 8 and y is 1" and returns 9, s is 9


# 2. Function with default arguments
def add(x=1, y=0):
    print(f"x is {x} and y is {y}")
    return x + y    # returns the result of x + y

# Calling function with default arguments
s = add()        # => prints out "x is 1 and y is 0" and returns 1, s is 1
s = add(5)       # => prints out "x is 5 and y is 0" and returns 5, s is 5
s = add(5,3)    # => prints out "x is 5 and y is 3" and returns 8, s is 8

# wrong way to define a function => SyntaxError: non-default argument follows default argument
# def my_function(a, b=5, c):
#     print(a, b, c)

#3.  Function that takes a variable number of positional arguments
def concatenate(*args):
   result = ''
   for tmp in args:
      result = result + tmp
   return result

# Calling the function
result = concatenate()
print(result)          # => '' -> empty string

result = concatenate('Python', '!')
print(result)          # => Python!

result = concatenate('I', 'love ', 'programming')
```

```
print(result)         # => Ilove programming


#4.  Function that takes a variable number of keyword arguments
def device_info(**kwargs):  #kwargs is a dictionary
   for k, v in kwargs.items():
       print(f'{k}: {v}')

# Calling the function
device_info(name='Cisco Router', ip='10.0.0.1', username='u1', password='secretpass')
# or:
d1 = {name='HP', ip='192.168.0.1', username='root', password='secret123'
device_info(**d1)
```

Scopes and Namespaces

```
x = 3    # this is a global scoped variable

def my_func1():
    print(f'x is {x}')  # this is "x" from the global namespace

# Calling the function
my_func1()      # => x is 3


def my_func2():
    x = 6                  # this is a local scoped variable
    print(f'x is {x}')    # this is NOT "x" from the global namespace

# Calling the function
my_func2()    # => x is 6
print(x)          # => 3 -> "x" variable was not modified inside the function


def my_func3():
    global x       # importing "x" from the global namespace
    x = x * 10    # this is "x" from the global namespace
    print(f'x is {x}')

# Calling the function
my_func3()    # => x is 30
print(x)          # => 30 -> global "x" variable was modified inside the function


def my_func4():
   print(f'x is {x}')
   x += 7      # this is an error, we used local x before assignment

## Calling the function
my_func4()      # => UnboundLocalError: local variable 'x' referenced before assignment
```

Lambda Expressions

```
# "x" and "y" are lambdas arguments.
add = lambda x, y: x + y    # this creates an anonymous function
type(add)                   # => function


# Assigning lambda expression to a variable
result = add(2, 3)    # => 5

# You can use default arguments
add = lambda x=1, y=0: x + y
result = add()    # => 1

# You can even use *args and **kwargs
my_function = lambda x, *args, **kwargs: (x, *args, {**kwargs})
# x is 2.3, args is (a, b, c) and kwargs is {arg1='abc', arg2='def', arg3='geh'}
my_function(2.3, 'a', 'b', 'c', arg1='abc', arg2='def', arg3='geh')


# Passing lambda as an argument to a function
# Lambdas are functions and can therefore be passed to any other function as an argument (or
returned from another function)
def my_func(x, fn):
   return fn(x)

result = my_func(2, lambda x: x**2)
print(result)    # => 4

result = my_func(2, lambda x: x**3)
print(result)     # => 8

result = my_func('a', lambda x: x * 3)
print(result)    # => 'aaa'

result = my_func('a:b:c', lambda x: x.split(':'))
print(result)    # => ['a', 'b', 'c'] -> this is a list
```

```
result = my_func(('p', 'y', 't', 'h', 'o', 'n'), lambda x: '-'.join(x))
print(result)    # => p-y-t-h-o-n > this is a string
```