Topics

- Functions
- Positional argument
- Name space
- Advanced functions
- Error handling

Functions

- Functions are an extremely important programming concept for structuring your code and avoiding repetitions

1. Keyword **def** marks the start of function header.
2. A function name to **uniquely identify** it. Function naming follows the same rule s of writing identifiers in Python.
3. Arguments through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (**usually 4 spaces**).
7. An optional **return statement** to return a value from the function.

```
def Hello():
    print('hello')
```

Functions

- Functions are an extremely important programming concept for structuring your code and avoiding repetitions

1. Keyword **def** marks the start of function header.
2. A function name to **uniquely identify** it. Function naming follows the same rule s of writing identifiers in Python.
3. Arguments through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (**usually 4 spaces**).
7. An optional **return statement** to return a value from the function.

```
def my_add(arg1, arg2):
    print(arg1+arg2)
    return arg1+arg2
```

Return values

- return the summation of arg1 and arg2
- return ‘True’ if the two arguments are equal

```
def my_add(arg1, arg2):
    return arg1+arg2
```

```
def my_add(arg1, arg2):
```

```
>>> my_add(3, 4)
>>>
```

Return values

- return the summation of arg1 and arg2
- return ‘True’ if the two arguments are equal

```
def my_add(arg1, arg2):
    return arg1+arg2
```

```
def my_add(arg1, arg2):
    if arg1 == arg2:
        return arg1+arg2, True
    else:
        return arg1+arg2, False
```

```
>>> my_add(3, 4)
>>> a, b = my_add(3, 4)
```

Positional Arguments

- The most familiar types of arguments are *positional arguments*, whose values are copied to their corresponding parameters in order.

```
def my_print(str1, str2):
    print(str2, str1)
```

```
>>> my_print('hello', 'student')
…
>>> my_print('hello', 'student', 'welcome')
…
```

```
def my_print(str1, str2, str3 = ''):
    print(str2, str1, str3)
```

Specify Default Parameter Values

The default is used if the caller does not provide a corresponding argument.

Positional Arguments

- The most familiar types of arguments are *positional argum ents*, whose values are copied to their corresponding para meters in order.

```
def menu(arg1, arg2, arg3):
    return {'wine': arg1, 'entree': arg2, 'dessert': arg3}
```

```
>>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
>>> menu('beef', 'bagel', 'bordeaux')
{'wine': 'beef', 'entree': 'bagel', 'dessert': 'bordeaux'}

>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'wine': 'bordeaux', 'dessert': 'bagel', 'entree': 'beef'}
>>> menu('frontenac', dessert='flan', entree='fish')
{'entree': 'fish', 'dessert': 'flan', 'wine': 'frontenac'}
```

Gather Positional Arguments with *

- An asterisk * groups variables into a tuple of parameter values

```
def print_args(*argments):
    print('Positional argument tuple:', argments)
```

```
>>> print_args()
Positional argument tuple: ()
```

```
def print_more(required1, required2, * argments):
    print(required1, required2, argments)
```

```
>>> print_more('cap', 'gloves', 'scarf', 'monocle','mustache wax')
cap gloves ('scarf', 'monocle', 'mustache wax')
```

Gather Positional Arguments with *

- An asterisk * is useful.

```
def my_add(arg1, arg2):
    return arg1+arg2
```

```
>>> my_add(3, 4)
7
```

```
>>> my_add(3, 4, 2, 4, 5, 6, 7, 8, 8, …)
…
```

```
def my_add(arg1, arg2, *arg3):
    return …
```

Docstrings

- You can attach documentation to a function definition by including a string at the beginning of the function body

```
def my_add(arg1, arg2):
    return arg1+arg2
```

```
def my_add(arg1, arg2):
    '''
    Adding two numbers
    >>> my_add(4, 5)
    '''
    return arg1+arg2
```

```
>>> help(my_add)
```

False, True, and None

- None is a special Python value that holds a place when there is nothing to say
- zero-valued integers or floats, empty strings (''), lists ([]), t uples **((,)),** dictionaries ({}), and sets(set()) are all false

```
>>> def do_nothing():
        pass

>>> a = do_nothing()
>>> type(a)
```

```
>>> def aa(arg1):
        if arg1 == True:
            print('True')
        elif arg1 == False:
            print('False')
        elif arg1 == None:
            print('None')
```

Exercise

- write a code to merge four mathematic functions into one function by using third argument

```
def my_add(arg1, arg2):
    return arg1+arg2
```

```
def my_mul(arg1, arg2):
    return arg1*arg2
```

```
def my_pow(arg1, arg2):
    return arg1**arg2
```

```
def my_div(arg1, arg2):
    return arg1/arg2
```

```
def my_cal(arg1, arg2, arg3):
    if arg3 == '+':
```

Exercise

- takes a variable (string, list, or tuple) and eliminate duplicated elements in the variable
- returns the variable as the same type of input variable

```
def mydup(c):
    tm = []
    for num in c:
        if num not in tm:
            tm.append(num)
    return tm
```

```
>>> mydup('ekeieislslkejjeite222')
ekislkjt2
>>> mydup(['a', 'a', 'b', '1', '1'])
['a', 'b', '1']
>>> mydup({'a', 'a', 'b', '1', '1‘})
{'a', 'b', '1‘}
```