# Functions

講者：Isaac

# Outline

- User-Defined Functions
- Built-in Functions

# User-Defined Functions

# User-Defined Functions

- Are reusable code blocks; write once, use multiple times.
- Modify per requirement.
- Well organized, easy to maintain.
- As user-defined functions can be written independently, the tasks of a project can be distributed for rapid application development.
- A well-defined and thoughtfully written user-defined function can ease the application development process.

# User-Defined Functions

- Rules for naming functions
  - It can begin with either of the following: A-Z, a-z, and underscore(_).
  - The rest of it can contain either of the following: A-Z, a-z, digits(0-9), and underscore(_).
  - A reserved keyword may not be chosen as an identifier.
- Define a function:

Definition

Indentation

```
>>> def hello():
        print('Bonjour~')
...
>>> hello()
Bonjour~
```

# User-Defined Function

- Syntax:

```
def <functionName>(<parameter1>[, <parameter2>…]):
    '''Docstring'''
    Code block
    return [value1, value2, …]
```

- Example:
  - Pass a parameter.

```
>>> def hello(name):
...     '''deliver name into the function'''
...     print('Hello~ ' + name)
...
>>> hello('Claire')
Hello~ Claire
```

Parameter

Pass

Argument

# Function Arguments

- Default Arguments:

```
def defaultArg(name, msg='Morning~'):
    print(msg + name)

defaultArg('Celine')
```

```
Morning~Celine
```

- Required Arguments:
  - Must be passed in correct number and order during function call

```
def requiredArg(name, grade):
    print(name + "'s math grade is: %s" %grade)

requiredArg('Marc', 98)
```

```
Marc's math grade is: 98
```

# Function Arguments

- Keyword Arguments:
  - keywords are mapped with the function arguments so the function can easily identify the corresponding values even if the order is not maintained during the function call.
  - Example:

```
def keywordArg(name, grade):
    print(name + "'s math grade is: %s" %grade)

keywordArg(name='Jack', grade=100)
```

```
Jack's math grade is: 100
```

```
keywordArg(grade=100, name='Jack')
```

```
Jack's math grade is: 100
```

# Function Arguments

- Variable number of arguments:
  - This is very useful when we do not know the exact number of arguments that will be passed to a function.
  - Example:

```
def varnumArg(*meals):
    for meal in meals: print(meal)

varnumArg('Pie', 'Ramen', 'Pasta')
```

```
Pie
Ramen
Pasta
```

# Function Arguments

- Variable number of arguments:
  - Use ** will pass a dictionary to the function.
  - Example:

```
def dinner(**kwargs):
    print('keyword arguments:', kwargs)

dinner(wine='cava', entree='lamb', dessert='tiramisu')
```

```
keyword arguments: {'wine': 'cava', 'entree': 'lamb', 'dessert': 'tiramisu'}
```

# User-Defined Function

- Return statement
  - Return the functions result to the caller.
  - Means the end of the functions.
  - Single/multiple value/object or none.
  - Example:

```
def returnExample():
    a = 'Hi'
    b = 'Hello'
    c = 'Hey'
    return a, b, c

returnExample()
```

```
('Hi', 'Hello', 'Hey')
```

# User-Defined Function

- Relations between Main program and User-defined functions.

Declaration

Pass

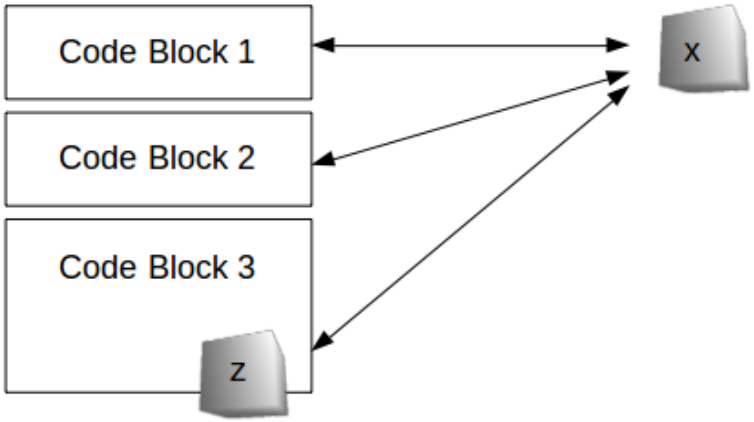
Program Entry point

```
def main():
    print('main function')
    subFunction(1,2)

def subFunction(par1, par2):
    print('parameters={},{}'.format(par1, par2))

if __name__=="__main__":
    main()
```

```
main function
parameters=1,2
```

# Variables between Functions

- There are two types of variables: **global variables** and **l ocal variables.**
- A global variable can be reached anywhere in the code, a l ocal only in the scope.




- A **global variable** (x) can be reached and modified anyw here in the code, **local variable** (z) exists only in block 3.

# Local variables

- Local variables can only be reached in their scope.

```
def sum(a, b):
    sum = a + b
    return sum

print(sum(1, 2))
print(x)
```

```
3

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-baaf5716f395> in <module>
      4
      5 print(sum(1, 2))
----> 6 print(x)

NameError: name 'x' is not defined
```

# Global Variables

- A global variable can be used anywhere in the code.

```
c = 99
def sum(a, b):
    global c
    sum = a + b + c
    print('inside the sum scope:{}'.format(c))
    return sum

print(sum(1, 2))
print('out of sum scope:{}'.format(c))
```

```
inside the sum scope:99
102
out of sum scope:99
```

# Variables between Functions

- Confusing Situation Example:
  - Please avoid naming same variable in different scopes.

```python
def lunch():
    waffle = 'lunch waffle'
    print(waffle)

def dinner():
    waffle = 'dinner waffle'
    print(waffle)
    lunch()
    print(waffle)

waffle = 'global'
dinner()
print(waffle)
```

```
dinner waffle
lunch waffle
dinner waffle
global
```

# Anonymous Function

- Lambda:
  - take any number of arguments with only one expression.
  - simply code
- Syntax:

```
lambda arguments : expression
```

- Example:

```python
def double(a):
    return a * 2

double(5)
```

```
10
```

```python
x = lambda a : a * 2

print(x(5))
```

```
10
```

# Syntax Candy

- Decorator:
  - takes another function and extends the behavior of the latter function without explicitly modifying it.
  - simply code

- Syntax:

```
@ function ( )
```

# Decorator

- Example

```
def print_func_name(func):
    def warp_1():
        print("Now use function '{}'".format(func.__name__))
        func()
    return warp_1

@print_func_name
def greeting():
    print("Good Morning !!!")

if __name__ == "__main__":
    greeting()

```

```
Now use function 'greeting'
Good Morning !!!
```

# Yield Expression

- Example:

```python
def square():
    for x in range(4):
        print(f'x inside square func:{x}')
        yield x ** 2

square_gen = square()

for x in square_gen:
    print(f'output:{x}\n')

```

```
x inside square func:0
output:0

x inside square func:1
output:1

x inside square func:2
output:4

x inside square func:3
output:9
```

# Built-in Functions

# Function

- Python function in any programming language is a sequence of statements in a certain order, given a name.
- When called, those statements are executed. So we don't have to write the code again and again for each [type of] data that we want to apply it to. This is called code re-usability.
- Call a function:

Argument

```
>>> abs(-1)
1
```

Return Value

# Common used built-in functions

```
abs(-1)
```

```
1
```

```
abs(1)
```

```
1
```

```
abs(0)
```

```
0
```

```
bool(0.5)
```

```
True
```

```
bool('')
```

```
False
```

```
bool(' ')
```

```
True
```

```
divmod(10,3)
```

```
(3, 1)
```

```
chr(65)
```

```
'A'
```

```
chr(97)
```

```
'a'
```

```
chr(9)
```

```
'\t'
```

```
chr(48)
```

```
'0'
```

```
# This Function takes a string as an argument,
# which is parsed as an expression.
x = 6
eval('x+6')
```

```
12
```

```
eval('x+(x%3)')
```

```
6
```

```
input('please enter a number:')
```

```
please enter a number:7
```

```
'7'
```

# Built-in Functions

| Built-in Functions | | | | |
|---|---|---|---|---|
| abs() | delattr() | hash() | memoryview() | set() |
| all() | dict() | help() | min() | setattr() |
| any() | dir() | hex() | next() | slice() |
| ascii() | divmod() | id() | object() | sorted() |
| bin() | enumerate() | input() | oct() | staticmethod() |
| bool() | eval() | int() | open() | str() |
| breakpoint() | exec() | isinstance() | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callable() | format() | len() | property() | type() |
| chr() | frozenset() | list() | range() | vars() |
| classmethod() | getattr() | locals() | repr() | zip() |
| compile() | globals() | map() | reversed() | __import__() |
| complex() | hasattr() | max() | round() | … |

# Built-in Functions

- Mathematics

```
abs(-1)
```

1

```
round(3.4)
```

3

```
pow(3, 2)
```

9

```
divmod(5, 2)
```

(2, 1)

```
lst = [1,2,3,4,5]
sum(lst)
```

15

```
min(lst)
```

1

```
max(lst)
```

5

# Built-in Functions

- Numeric

```
1 txt='100'
2 a = int(txt)
3 print(f'a = {a}, {type(a)}')
```

```
a=100, <class 'int'>
```

```
1 txt='100.11'
2 a = float(txt)
3 print(f'a = {a}, {type(a)}')
```

```
a=100.11, <class 'float'>
```

```
1 str(10)
```

```
'10'
```

```
1 bin(10)
```

```
'0b1010'
```

```
1 oct(10)
```

```
'0o12'
```

```
1 hex(10)
2
```

```
'0xa'
```

# Built-in Functions

- Numeric/Data Structure

```python
chr(97)
```

```
'a'
```

```python
chr(0x30)
```

```
'0'
```

```python
print(bool(0))
print(bool(1))
print(bool(2))
```

```
False
True
True
```

```python
lst = ['1','2','3','4']
a = list(map(int,lst))

print(f'{a}, {type(a)}')
```

```
[1, 2, 3, 4], <class 'list'>
```

```python
def f(x):
    return x**2

lst =[1,2,3,4]

list(map(f,lst))
```

```
[1, 4, 9, 16]
```

```python
lst =[1,2,3,4]
list(map(lambda x: x**2, lst))
```

```
[1, 4, 9, 16]
```

# Built-in Functions

- Numeric/Data Structure

```
lst =[1,2,3,4]
list(filter(lambda x: x>2, lst))
```

```
[3, 4]
```

```
lst1 = [1, 2, 3, 4]
lst2 = ['a', 'b', 'c', 'd']
zip_l1_l2 = zip(lst1, lst2)

list(zip_l1_l2)
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

```
lst1 = [1, 2, 3, 4]
lst2 = ['a', 'b', 'c', 'd']
zip_l1_l2 = zip(lst1,lst2)

dict(zip_l1_l2)
```

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

# Built-in Functions

- Numeric/Data Structure

```
a = bytes([1,2,3,4])
print(f'{a}, {type(a)}')
```

```
b'\x01\x02\x03\x04', <class 'bytes'>
```

```
a = bytes('hello','ascii')
print(f'{a}, {type(a)}')
```

```
b'hello', <class 'bytes'>
```

```
a = bytearray([1,2,3])
print(f'{a}, {type(a)}')
```

```
bytearray(b'\x01\x02\x03'), <class 'bytearray'>
```

```
a = bytearray('hello', 'utf-8')
print(f'{a}, {type(a)}')
```

```
bytearray(b'hello'), <class 'bytearray'>
```

# Built-in Functions

- all()

```python
s = {0: 'False', 1: 'False'}
print(all(s))

s = {1: 'True', 2: 'True'}
print(all(s))

s = {1: 'True', False: 0}
print(all(s))

s = {'0': 'True'}
print(all(s))
```

```
False
True
True
True
```

```python
# all values true
l = [1, 2, 3, 4]
print(all(l))

# all values false
l = [0, False]
print(all(l))

# one false value
l = [1, 0, 3, 4]
print(all(l))

# one true value
l = [0, False, 1]
print(all(l))
```

```
True
False
True
False
```

# Built-in Functions

- any()

```python
s = {0: 'False', 1: 'False'}
print(any(s))

s = {1: 'True', 2: 'True'}
print(any(s))

s = {1: 'True', False: 0}
print(any(s))

s = {'0': 'True'}
print(any(s))
```

```
True
True
True
True
```

```python
# all values true
l = [1, 2, 3, 4]
print(any(l))

# all values false
l = [0, False]
print(any(l))

# one false value
l = [1, 0, 3, 4]
print(any(l))

# one true value
l = [0, False, 1]
print(any(l))
```

```
True
False
True
True
```

# Built-in Functions

- iter() & next()

```python
txt = ['p', 'y', 't', 'h', 'o', 'n']
txt_iter = iter(txt)
print(txt_iter)

print(next(txt_iter))
print(next(txt_iter))
print(next(txt_iter))
print(next(txt_iter))
print(next(txt_iter))
print(next(txt_iter))
```

```
<list_iterator object at 0x105c89550>
p
y
t
h
o
n
```

# Built-in Functions

- eval()

```python
# eval():
# parse string and runs in python code
print(eval('pow(2,2)'))
print(eval('2 + 1'))

x = "[[1,2], [3,4], [5,6], [7,8], [9,0]]"
y = eval(x)
print(f'{y}, {type(y)}')

x = "{1:'xx',2:'yy'}"
y = eval(x)
print(f'{y}, {type(y)}')

x = "(1,2,3,4)"
y = eval(x)
print(f'{y}, {type(y)}')

```

```
4
3
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]], <class 'list'>
{1: 'xx', 2: 'yy'}, <class 'dict'>
(1, 2, 3, 4), <class 'tuple'>
```

# Built-in Functions

- exec()

```
# exec()
program = 'x = 2\ny=3\nprint("Sum =", x+y)'
exec(program)
```

```
Sum = 5
```