SI100B/F: Introduction to Information

Science and Technology

Part I: Python Programming

殷树

School of Information Science and Technology

ShanghaiTech University

Outline

More on Loops

- the `else` statement together with loops
  - `while else` structure
  - `for else` structure

Functions

- block of codes to implement a specific function
  - definition
  - `return` statement in function
  - calling
  - arguments passing

The Else Clause

The `else` clause can also be used together with the `while` or `for` loop, which is unique to Python

syntax

```
while <expr> :
   statements
else :
   statements
```

or

```
for <var> in <iterable> :
   statements
else :
   statements
```

the `else` clause

- is executed
  - `expr` turns `False` in the `while` loop
  - exhaustion of the iterable in the `for` loop
- is not executed
  - when the loop is terminated using `break`

In [1]:
```
# returns the prime numbers between 2 and 20

import math
i = 2
while i < 100 :
    j = 2
    while j <= math.sqrt(i) :
        if not(i%j) :
            break
        j = j + 1
    # the else part is executed when the above while loop exhausts
    else :
        print(i, 'is prime')
    i = i + 1
```

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

In [2]:
```
for n in range(2, 20) :
    for x in range(2, n) :
        if n % x == 0 :
            print( n, 'equals', x, '*', int(n/x))
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Function

- math: a map between input and output (math)
- programming: self-contained block of code that encapsulates a specific task or related group of tasks
- provide better modularity for application and a high degree of code reusing

Two types of functions

- built-in Python functions
  - `len()`, `type()`, `input()`, `any()`, et. al.
- user defined functions

Synonyms in other languages

- subroutines
- procedures
- methods
- subprograms

How to Define a Function

Syntax

```
def <function_name>(<parameters>) :
    statements
    ......
```

Component	Meaning
`def`	keyword that informs Python that a function is being defined
`<function_name>`	valid Python identifier that names the function
`<parameters>`	an optional, comma-separated list of parameters that may be passed to the function
:	punctuation that denotes the end of the Python function header
`statements`	block of valid Python statements

- `statements` part is called the body of the function

In [3]:
```
# define a function that determine whether a number is prime or not

import sys
def is_prime(a) :
    if type(a) is not int :
        sys.exit('you have to input an int')
    elif a < 2 :
        sys.exit('you have to input a positive int not less than 2')
    else :
        for i in range(2, a) :
            if not a % i :
                print(a, 'is not prime')
                break
        else :
            print(a, 'is prime')
```

Function Calls

Syntax

```
<function_name>(<arguments>)
```

- `<arguments>` are the values passed into the function
  - they correspond to the `<parameters>` in the function definition
- functions can also take no arguments
  - while calling such a function, parentheses are required
  - parentheses are required to define such a function

In [4]:
```
# check whether 23 is a prime number

is_prime(223)
```

```
223 is prime
```

In [5]:
```
is_prime(21)
```

```
21 is not prime
```

In [6]:
```
# define a function that takes no arguments

def f() :
    s = 'just print something'
    print(s)

# now call such a function
f()
```

```
just print something
```

The Return Statement

1. immediately terminates the function and passes execution control back to the caller

   Example: the `is_prime` function

```
def if_prime(x) :
  if type(x) != int :
      print('please pass an int type number')
      return
  elif x <= 0 :
      print('the input should be positive')
      return
  elif x == 1 or 2 :
      print('whether', x, 'is prime or not is decided here')
      return
  else :
      pass    # did nothing, just place holding to avoid syntax error
  print('finished')
```

In [7]:
```
def if_prime(x) :
    if type(x) != int :
        print('please pass an int type number')
        return
    elif x <= 0 :
        print('the input should be positive')
        return
    elif x == 1 or x == 2 :
        print('whether', x, 'is prime or not is decided here')
        return
    else :
        pass    # the pass statement did nothing,just holds this place to avoid syntax error
    print('finished')
```

In [8]:
```
if_prime(-3)
```

```
the input should be positive
```

In [9]:
```
if_prime(21)
```

```
finished
```

The Return Statement

2. pass data back to the caller

- any types can be passed back

  For example, `feval` returns the value of the function $x^3 + 5x^2 + 4x - 1$ for given x

  ```
  def feval(x) :
      return x**3 + 5*x**2 + 4*x -1
  ```

- while returning multiple comma-separated expressions, Python packs them as a tuple, e.g.,

  ```
  def f() :
    return 'first_it', 'second_it', 'third_it'
  ```

In [10]:
```
# return numbers
def feval(x) :
    return x**3 + 5*x**2 + 4*x -1
```

In [11]:
```
feval(32)
```

Out[11]:
```
38015
```

In [12]:
```
def f() :
    return 'first', 'second', 'third'
```

In [13]:
```
print(type(f()))
print(f())
```

```
<class 'tuple'>
('first', 'second', 'third')
```

More on Function Return

1. when no return value is given, a Python function returns the keyword `None`, e.g.,

   ```
   def feval_1() :
     return
   ```

2. if the function body does not contain a return statement at all, the function also returns the keyword `None`, e.g.,

```
    def print_info() :
        print('just for fun!')
```

- recall that `None` is `False` when evaluated in a boolean context

In [14]:
```
# return without expression following, Python returns 'None'

def feval_1() :
    return

x = feval_1()
print(x)
```

```
None
```

In [15]:
```
# function without return statement, Python also returns 'None'
def print_info() :
    pass

x = print_info()
print(x)
```

```
None
```

Argument Passing

- arguments: data we pass into a function while calling
  - e.g., `is_prime(37)`, `37` is the argument that we pass into the function
- arguments are put in the parentheses following the function name
- some functions take no arguments
  - e.g., `f()` takes no arguments, but still performs some task

Parameters and **Arguments**

- parameters are used in the function definition
  - abstract, no specific value (often)
- arguments are the data passed to call a function

Four types of function arguments

- required arguments (positional arguments)
- keyword arguments
- default arguments
- variable-length arguments

Required Arguments

The

- **order/position**
- **number**

of arguments passed into the function should match exactly the parameters in the function definition.

For example

```
    def print_info(name, age) :
        print('The name is', name, 'and the age is', age)
```

while calling `print_info()`, you have to

- pass two arguments
  - 'Tom', 21, and use parentheses to seperate
- order the arguments correctly
  - 'Tom' --> `name`, 21 --> `age`

In [16]:
```
# illustration of required arguments
def print_info(name, age) :
        print('The name is', name, 'and the age is', age)

# call the function
print_info('Tom', 21)
```

```
The name is Tom and the age is 21
```

In [17]:
```
# pass no argument
print_info()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1548390/2634305884.py in <module>
      1 # pass no argument
----> 2 print_info()

TypeError: print_info() missing 2 required positional arguments: 'name' and 'age'
```

In [18]:
```
# pass one argument
print_info('Tom')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1548390/522848603.py in <module>
      1 # pass one argument
----> 2 print_info('Tom')

TypeError: print_info() missing 1 required positional argument: 'age'
```

In [19]:
```
# mix the arguments order
print_info(21, 'Tom')
```

```
The name is 21 and the age is Tom
```

Keyword Argument

when using keyword arguments to call a function

- the caller identifies the arguments by the parameter name

e.g,

```
print_info(name = 'Tom', age = 21)
```

or

```
print_info(age = 21, name = 'Tom')
```

or even

```
print_info('Tom', age = 21)
```

however,

```
print_info(name = 'Tom', 21)
```

is not allowed. Positional arguments cannot follow keyword arguments.

In [20]:
```
print_info('Tom', age=21)
```

```
The name is Tom and the age is 21
```

In [21]:
```
print_info(name = 'Tom', age = 21)
```

```
The name is Tom and the age is 21
```

In [22]:
```
print_info(age = 21, name = 'Tom')
```

```
The name is Tom and the age is 21
```

In [23]:
```
print_info(name = 'Tom', 21)
```

```
  File "/tmp/ipykernel_1548390/3491764136.py", line 1
    print_info(name = 'Tom', 21)
                             ^
SyntaxError: positional argument follows keyword argument
```

Default Arguments

a default argument is

- assumed a default value if a value is not provided in the function call for that argument

e.g., define the following function

```
    def print_more(name, age, gender='male') :
        print('name is', name)
        print('age is', age)
        print('gender is', gender)
```

while calling, we can use

```
print_more('Tom', 21)
```

or

```
print_more('Laura', 21, 'female')
```

or

```
print_more('Laura', gender='female', age = 21)
```

In [24]:
```
# illustration of default arguments

def print_more(name, age, gender='male') :
    print('name is', name)
    print('age is', age)
    print('gender is', gender)
```

In [25]:
```
print_more('Tom', 21)
```

```
name is Tom
age is 21
gender is male
```

In [26]:
```
print_more('Laura', 21, 'female')
```

```
name is Laura
age is 21
gender is female
```

In [27]:
```
print_more('Laura', gender='female', age = 21)
```

```
name is Laura
age is 21
gender is female
```

Variable-length Arguments

used for functions that we donot know the number of arguments to take

Syntax

```
def function_name(*args) :
    statements
```

- use the symbol ***** before a variable name, such as `*args` to hold the values of all arguments used in the function
- arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter `args` excluding the asterisk *****
- *args is used to send a non-keyworded variable length argument list to the function
- `args` is just a variable name, not a keyword

In [28]:
```
# compute the arithmetic mean
def arithmetic_mean(*args) :
    print(type(args))
    print(args)
    return sum(args)/len(args)

print(arithmetic_mean(1, 2))
```

```
<class 'tuple'>
(1, 2)
1.5
```

In [29]:
```
print(arithmetic_mean(1, 2, 3))
```

```
<class 'tuple'>
(1, 2, 3)
2.0
```

In [30]:
```
x = [1, 2, 3, 4]
print(arithmetic_mean(*x))
```

```
<class 'tuple'>
(1, 2, 3, 4)
2.5
```

Keyworded Variable-length Arguments

syntax

```
python    def function_name(**kwargs) :        statements
```

- **kwargs allows to pass keyworded variable length of arguments to a function
- arguments passed to the function are of dict type, and these passed arguments make a dict inside the function with same name as the parameter `kwargs` excluding the symbol **
- `kwargs` is just a variable name, not a keyword

In [31]:
```
# demonstration of **kwargs

def print_pinfo(**kwargs) :
    print(type(kwargs))
    for key, value in kwargs.items() :
        print("{} is {}".format(key,value))

print_pinfo(Firstname="Tim", Lastname="Duncan", Age=22, Phone=1234567890)
```

```
<class 'dict'>
Firstname is Tim
Lastname is Duncan
Age is 22
Phone is 1234567890
```

In [32]:
```
print_pinfo(Firstname="Donald", Lastname="Trump", Age=74, Phone=1234567890, email="stupid_guy@idiot.com")
```

```
<class 'dict'>
Firstname is Donald
Lastname is Trump
Age is 74
Phone is 1234567890
email is stupid_guy@idiot.com
```