SI100B/F: Introduction to Information

Science and Technology

Part I: Python Programming

殷树

School of Information Science and Technology

ShanghaiTech University

Outline

Python Program Structure

- Sequential flow
- Conditional flow
- Repeated flow

More on Loops

- iteration means executing the same block of code over and over
- a program structure that implements iteration is called a loop
  - `while` statement
  - `for` statement

Functions

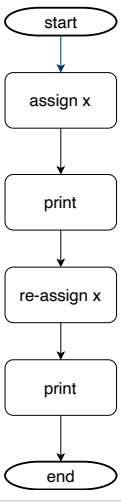
- block of codes to implement a specific function

1. Sequential flow

A program is a collection of instructions to be executed

- executed in exact top-down order

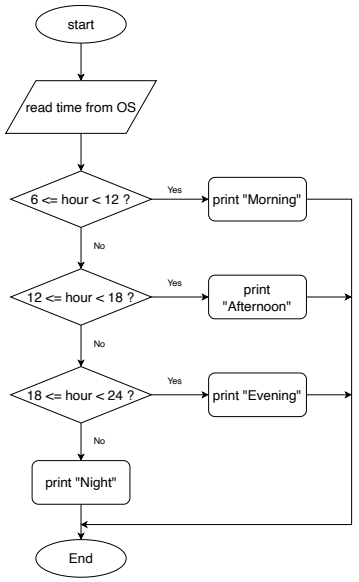
```
x = 2
print(x)
x = x + 2
print(x)
```



2. Conditional flow

- execute only when conditions are satisfied
  - often `if...else...` statement used
    - indentation is key to locate the branch
  - code structure

```
# a typical branch
if expr1 :
  excute block 1
elif expr2 :
  excute block 2
elif expr3 :
  excute block 3
else :
  excute block x
print('\nFinished')
```



In [1]:
```
import time
# get the current hour from OS
t = time.localtime().tm_hour
# check the time range
if 6 <= t < 12 :
    print('Morning')
elif 12 <= t < 18 :
    print('Afternoon')
elif 18 <= t < 24 :
    print('Evening')
else :
    print('Night')
print('\nFinished')
```

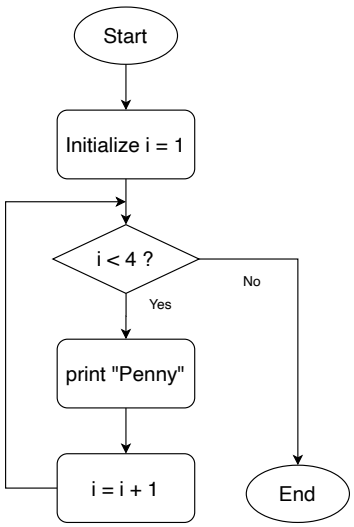
```
Evening

Finished
```

3. Repeated flow

- execute code block multiple times
  - often used with `while` , and `for` statements
- e.g., print "Penny" three times

```
i = 1
while i < 4 :
  print("Penny")
  i = i + 1
print('\nFinished')
```

- loops : code blocks that are executed multiple times
  - an index that change each time (i in this example) to stop the repeatition



while Statement (Indefinite Iteration)

- code block executes until some condition is met

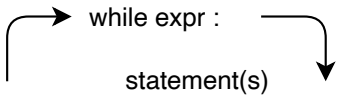
Syntax

```
while expr :
    statement(s)
```

- `expr` is of Boolean type
  - `True` : `statements` will be executed
  - `False` : `statements` will not be executed
- `statement(s)` represents the block to be repeatedly executed
  - often referred to as the body of the loop
  - denoted with indentation

`while` loop

- `expr` is first evaluated, if `True` , the body will be executed
- `expr` will be evaluated again in the next iteration
- terminated when `expr` becomes `False`



In [2]:
```
x = 3
while x > 0 :
    print(x)
    x -= 1
print('Ready, Go!')
```

```
3
2
1
Ready, Go!
```

while Loop Termination

Loop shall be excuted finite times

- controlled via `expr`
- `break` can also be used (later this lecture)

What will happen?

```
n = 5
while n > 0 :
    print('Aha')
```

- `n` is untouched in the iteration
- condition `n > 0` is always `True`
- the `while` loop will be executed infinite times

List Type Involved in while Loop

When a list is evaluated in the Boolean context

- `True` if it has elements
- `False` if it is empty

In [3]:
```
a = ['foot', 'ball', 'world', 'cup']
while a :
    print(a.pop(0))
print('a == []:', a == [])
```

```
foot
ball
world
cup
a == []: True
```

Nested while Loop

Nested Loop

- one loop inside another loop

Syntax

```
while expr1 :
     statements
     ......
     while expr2 :
        statements
        ......
     statements
     ......
```

- scope of the loop is determined by indentation
- you can put any type of loop inside any other type of loop, e.g.,
  - a `for` loop inside a `while` loop

- a `while` loop inside a `for` loop

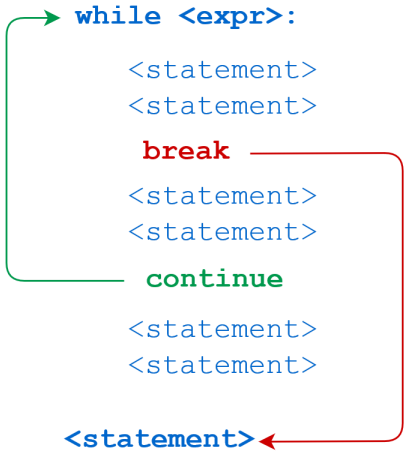
In [4]:
```
# a simple nested while loop example

a = ['some', 'any']
while a :
    print(a.pop(0))
    b = ['one', 'thing']
    while b :
        print('-', b.pop(0))
```

```
some
- one
- thing
any
- one
- thing
```

Loop Termination

To terminate a loop prematurely, we can use

- `break` statement, to immediately terminate a loop entirely. Program execution proceeds to the first statement following the loop body.
- `continue` statement, to immediately terminate the current loop. Execution jumps to the top of the loop, and the controlling expression is re-evaluated.



In [5]:
```
# demonstrate how break works

x = 5
while x > 0 :
    x = x - 1
    if x == 2 :
        break
    print(x)
print('Loop ends while x = ', x)
```

```
4
3
Loop ends while x =  2
```

In [6]:
```
# demonstrates how continue works

x = 5
while x > 0 :
    x = x - 1
    if x == 2 :
        continue
    print(x)
print('Loop ends while x = ', x)
```

```
4
3
1
0
Loop ends while x =  0
```

break or continue in Nested Loops

A `break` or `continue` statement found within nested loops applies to the nearest enclosing loop

```
while expr1 :
    statement
    statement
    ......

    while expr2 :
        statement
        statement
        break    # break applies to the <while expr2> loop

    break    # break here applies to the <while expr1> loop
    statement
```

In [7]:
```
# find prime numbers from 2 to 100

import math
i = 2
while i < 100 :
    j = 2
    while j < math.sqrt(i) :
        if not(i%j) :
            break
        j = j + 1
    if j > math.sqrt(i) :
        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
```

for Statement (Definite Iteration)

- number of repetitions is specified explicitly in advance

Syntax

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

- `<iterable>` ; is a collection of objects, e.g., a list, tuple, or dict
- `statements` are denoted by indentation, and are executed once for each item in `<iterable>`
- `<var>` ; takes on the value of the next element in `<iterable>` each time through the loop

In [8]:
```
# for loop using a list as an iterable

a = ['something', 'like', 'this']
for i in a :
    print(i)
```

```
something
like
this
```

In [9]:
```
# for loop using a tuple as an iterable

a = ('foo', 'bar', 'orz')
for i in a :
    print(i)
```

```
foo
bar
orz
```

In [10]:
```
# for loop through a dict

a = {'foo': 1 , 'bar': 2 , 'orz': 3}
for i in a :
    print(i)
```

```
foo
bar
orz
```

In [11]:
```
# access dict values within for loop

a = {'foo': 1 , 'bar': 2 , 'orz': 3}
for i in a :
    print(a[i])
```

```
1
2
3
```

In [12]:
```
# or direct use .values()

a = {'foo': 1 , 'bar': 2 , 'orz': 3}
for i in a.values() :
    print(i)
```

```
1
2
3
```

In [13]:
```
# access the key and value of a dict simultaneously

a = {'foo': 1 , 'bar': 2 , 'orz': 3}
for i, j in a.items() :
    print('i = ', i, ', j = ', j)
```

```
i =  foo , j =  1
i =  bar , j =  2
i =  orz , j =  3
```

Iterable

Recall the syntax

```
for <var> in <iterable>
    statements
    ......
```

In Python, **iterable** means an object can be used in iteration. It can be used as

- an adjective: an object can be described as iterable
- a noun: an object can be characterized as an iterable

`iter()` built-in function

- applied to an iterable
- returned something called **iterator**

`next()` built-in function

- used to obtain the next value from an iterator

In [14]:
```
a = 'some'
iter(a)    # string as an iterable
```

Out[14]:
```
<str_iterator at 0x7fdf90365e10>
```

In [15]:
```
li = ['this', 'is', 'a', 'list']
iter(li)    # list as an iterable
```

Out[15]:
```
<list_iterator at 0x7fdf90365050>
```

In [16]:
```
tu =('a', 'tuple', 'example')
iter(tu)    # tuple as an iterable
```

Out[16]:
```
<tuple_iterator at 0x7fdf90373390>
```

In [17]:
```
li = ['just', 'an', 'example']
itr = iter(li)
print(next(itr))
print(next(itr))
print(next(itr))
```

```
just
an
example
```

Iterable Types

types that are **iterable**

- string
- list
- tuple
- dict
- set

types that are **not iterable**

- int
- float
- built-in function

In [18]:
```
a = 42
iter(a)    # int is not iterable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1076536/2301834340.py in <module>
      1 a = 42
----> 2 iter(a)    # int is not iterable

TypeError: 'int' object is not iterable
```

In [19]:
```
iter(len)    # built-in function is not iterable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1076536/4015810181.py in <module>
----> 1 iter(len)    # built-in function is not iterable

TypeError: 'builtin_function_or_method' object is not iterable
```

Iterable Using the Range Function

- Python `range` type generates a sequence of integers by defining a start and the end point

```
range(start, stop, step)    # an integer sequence from start to (stop -1) with step size `step`

range(start, stop)    # an integer sequence from start to (stop -1) with step size `1`
```

  - The stop argument must be greater than start. Otherwise, the sequence is empty:

```
range(stop)    # an integer sequence from 0 to (stop -1) with step size `1`
```

  - If the argument is 0 or a negative integer, range returns an empty sequence
- It is generally used with the for loop to iterate over a sequence of numbers.

In [20]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [21]:
```
for i in range(3, 5):
    print(i)
```

```
3
4
```

In [22]:
```
for i in range(20, 4, -5):
    print(i)
```

```
20
15
10
5
```

In [25]:
```
for i in range(3, 1, -1):
    print(i)
```

```
3
2
```

For Loop Summary

relavant terms

Term	Meaning
iteration	looping through the objects or items in a collection
iterable	an object (or the adjective used to describe an object) that can be iterated over
iterator	the object that produces successive items or values from its associated iterable
`iter()`	built-in function used to obtain an iterator from an iterable
`next()`	built-in function to get the next value from an iterator

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

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 [26]:
```
import math
i = 2
while i < 20 :
    j = 2
    while j <= math.sqrt(i) :
        if not(i%j) :
            break
        j = j + 1
    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
```

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 (part in this lecture)

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 [27]:
```
# 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 < 0 :
        sys.exit('you have to input a positive int')
    else :
        for i in range(2, a) :
            if not a % i :
                print(a, 'is not prime')
                break
        if i == a - 1 :
            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 [28]:
```
is_prime(21)
```

```
21 is not prime
```