2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 1-2: Control flow

Kai-Feng Chen
National Taiwan University

CONTROL FLOW

- You are not always on the same route, do you?

CONDITIONAL EXECUTION

- One definitely needs the ability to check some certain conditions and change the behavior of the program accordingly.
- Conditional statements give us this ability.
- The simplest form is the **if statement**:

```
if x > 0:
    print('x is positive!')
```

- The **boolean expression** (`x>0`) after if is called the **condition**.
- If it is true, then the indented statement (print) gets executed. Otherwise nothing will happen.

THE STRUCTURE

The condition, which must be a boolean expression.

header is ending with a **colon (:)**

```
if x > 0:
    print('x is positive!')
```

HEADER

BODY

Body has to be indented.

The body can contain any number of statements.

Indentation is important. The lines started with the same indentation is treated as a group. The convention is four spaces, but this is not restricted.

BOOLEAN EXPRESSIONS

- A boolean expression is either true or false.
- For example, the basic operator "==" compares two operands and produces **True** if they are equal:

```
>>> 5==5
True
>>> 5==6
False
```

- The **True** and **False** (**T**,**F** must be capital) belong to bool type:

```
>>> type(True)
<class 'bool'>
```

RELATIONAL OPERATORS

- The == operator is one of the relational operators; the others are:

```
x != y      # x is not equal to y
x > y       # x is greater than y
x < y       # x is less than y
x >= y      # x is greater than or equal to y
x <= y      # x is less than or equal to y
```

- A common error is to use a single **equal sign "="** instead of a **double equal sign "=="**.
- There is no "=<" or "=>".

LOGICAL OPERATORS

- There are three logical operators: "**and**", "**or**", "**not**".
- The operands of the logical operators should be boolean expressions, but Python is not very strict – any nonzero number is interpreted as True.

```
x > 0 and x < 10
```
➡ True 0 < x < 10
➡ False otherwise

```
n % 2 == 0 or n % 3 == 0
```
➡ True if n is multiple of 2 or 3
➡ False otherwise

```
not (x > y)
```
➡ True if x is less or equal to y
➡ False x > y

ALTERNATIVE EXECUTION

- Alternative execution — with two possibilities and the condition determines which one gets executed.
- The syntax:

```
if x%2 == 0:
    print('x is even')
else:
    print('x is odd')
```

- Since the condition must be true or false, exactly one of the **branches** will be executed.

CHAINED CONDITIONALS

- Sometimes there are more than two possibilities — one way to express a computation like that is a chained conditional:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

- Exactly one branch will be executed. There is no limit on the number of **elif** (*"else if"*) statements.
- If there is an **else**, it has to be at the end.

NESTED CONDITIONALS

- One conditional can also be nested within another. For example:

If-else block #1

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

If-else block #2

indentation of the statements define the blocks.

The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own.

NESTED CONDITIONALS (II)

- Indentation makes the structure apparent. But nested conditionals become difficult to read very quickly. Good to avoid them.
- **Logical operators** often provide a way to simplify nested conditional statements.

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

Simplified with "and":

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

COMMENT: INDENTATION

- Leading whitespace at the beginning of lines is used to define the indentation level of the line, which is used to determine the **grouping of statements**.
- Due to the nature of various text editors, it is unwise to use a mixture of spaces and tabs for the indentation.

Example of a correctly (but confusingly?) indented code:

```
def perm(l):
    if len(l) <= 1:
        return [l]
    r = []
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(s)
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

INTERMISSION

- Try the following logic operations:

```
>>> a = 1.0/3.0
>>> b = 1.0 - 2.0/3.0
>>> a == b
```

Do you see **True** or **False**? Why?

```
>>> not (((True and (True or False) and True) and
False) or False) and False or True
```

Do you see **True** or **False**? Why?

INTERMISSION (II)

- Try this kind of “weird-indented” programs?

```
  print "a"
    print "aa"
      print "aaa"
        print "aaaa"
```

or

```
x = 3
if x < 4:
    print('x is smaller than 4.')
  if x > 1:
      print('x is greater than 1.')
```

Which line do you expect to see the (syntax) error?

INTERMISSION (III)

- Try this kind of “weird-indented” programs?

```
x = y = 2
if x > 1:
    if y < 4:
        x += y
        y += 4
    elif x + y > 1:
        y = (x + y)*2
    else:
        x -= (y / 2)
elif y == x % 2:
    y = x**2
else:
    x -= y**2
print(x,y)
```

⇐ **x += y** is the same as **x = x + y**
y += 4 is the same as **y = y + 4**

What do you expect to see in the end (x,y)?

ITERATION

- Computers are often used to automate repetitive tasks: **repeating identical or similar tasks**.
- Because iteration is so common, Python provides several language features to make it easier:
  - The **while** statement, which can be used as **general loops**.
  - The **for** statement, which is usually used to run through a block of code for **each item in a list** sequentially.
- An example (printing 1 to 10 on the screen):

```
n = 1
while n <= 10:
    print(n)
    n += 1
```

THE "WHILE" STATEMENT

Setting the initial value to n

The condition

header is also ending with a **colon**

```
n = 1
while n <= 10 :
    print(n)
    n += 1
```

HEADER

BODY

n += 1 is the same as **n = n + 1**, means **n** should be updated with **n+1**

- The flow of execution for a **while** statement:
  - Evaluate the condition, yielding **True** or **False**.
  - If the condition is false, exit the while statement.
  - If the condition is true, execute the body and then go back to step 1.

TERMINATION OF THE LOOP

- The loop should change some of the variables so that eventually the condition becomes false and ends the loop (e.g. the **n+=**1 statement).
- Otherwise the loop will repeat forever as **an infinite loop**.
- Or use the **break** statement to jump out of the loop:

```
n = 1
while True:  ⇐ an infinite loop actually!
    print(n)
    if n>=10:
        break  ⇐ jump out of the loop
    n += 1
```

CONTINUE THE LOOP

- *Break* versus *Continue* (can be confusing for beginners):
  - The **break** statement, which should jump out of the loop can continue to execute the next statement.
  - The **continue** statement, which should jump back to the head of the loop, check the condition, and continue to the next iteration.

```
n = 0
while True:
    n += 1
    if n > 10:
        break
    if n % 2 == 0:
        continue
    print(n)
print('the end!')
```

continue the loop

break the loop

LOOP – ELSE STATEMENT

- Similar to the if statement, **else** can be attached to the end of loop.
- The code block after else will be executed if the loop ended normally (without calling the **break** statement):

```
n = 0
m = 5  ⇐ if m is not between 1–10, the 'ended normally!' will be printed.
while n<10:
    n += 1
    print n
    if n==m:
        print('n ==',m,'hit. Break the loop.')
        break
else:
    print('ended normally!')
```

THE "PASS" STATEMENT

- The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.
- For example:

```
while True:
    pass # this is a do-nothing, infinite loop
```

```
n = 3
if   n == 1:
    print('One!')
elif n == 2:
    print('Two!')
elif n == 3:
    print('Three!')
else:
    pass  ⇐ do nothing, simply a placeholder for now
```

THE "FOR" STATEMENT

- The **for** statement in Python differs a bit from what you may be used to in C – it iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.
- An example (iterating the letters in a string):

```
letters = 'bhJlmprsty'
for initial in letters:   ⇐ loop over the letters
    print(initial + 'ack', end=' ')
```

Output:

```
back hack Jack lack mack pack rack sack tack yack
```

ACCESSING A LIST

- The **for** statement can access to the elements in a list. For example:

```
animals = ['cat','dog','horse','rabbit','mouse']
for a in animals:
    print(a)

odd_numbers = [1,3,5,7,9]
for n in odd_numbers:
    print(n)
```

The list "odd_number" can be actually replaced by a simple call to the built-in function **range()**

THE RANGE() FUNCTION

- If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It can generate lists containing arithmetic progressions:

```
>>> range(10)
range(0, 10)
>>> type(range(10)) ⇐ since python 3 range() has its own type!
<class 'range'>
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

The given end point (e.g. 10) is never part of the generated list.

RANGE() + FOR

- Get a list of odd numbers:

```
for n in range(1,10,2):
    print(n)
```

- To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
animals = ['cat','dog','horse','rabbit','mouse']
for i in range(len(animals)):
    print(i,'=>',animals[i])
```

The function **len()** return the number of elements in a list or how many characters in a string.

A LITTLE BIT MORE ON THE LIST

- Python has a number of data types, used to group together other values. The most versatile is the **list**. For example:

```
>>> a = ['spam','eggs',100,1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]   ⇐ Access to the elements in a reversed order
100
>>> a[1:3]   ⇐ Get a sub list
['eggs', 100]
```

A LITTLE BIT MORE ON THE LIST (CONT.)

- Few more operations with python list:

```
>>> b = ['spam','eggs',[1,2,[3,4,5]]] ⇐ anything can be in a list;
>>> b                                   including another list!
['spam', 'eggs', [1, 2, [3, 4, 5]]]
>>> b[0] = 'foo' ⇐ list is mutable!
>>> b
['foo', 'eggs', [1, 2, [3, 4, 5]]]
>>> b.append(1+3j)
>>> b
['foo', 'eggs', [1, 2, [3, 4, 5]], (1+3j)]
>>> len(b) ⇐ there are 4 items!
4
```

A LITTLE BIT MORE ON THE STRING

- The basic operations on the string are quite similar to that of list.
- A similar way can be used to access individual characters, or a sub-string.

```
>>> word = 'Help' + 'Me'
>>> word
'HelpMe'
>>> word[4]
'M'
>>> word[0:2]  ⇐ Get a sub string
'He'
>>> word[0:-2]
'Help'
>>> len(word) ⇐ character counting
6
```

A LITTLE BIT MORE ON THE STRING (CONT.)

- Few more operations with python string:

```
>>> fruit = 'banana'
>>> fruit
'banana'
>>> fruit.upper()
'BANANA'  ⇐ return the upper case
>>> fruit[0]
'b'
>>> fruit[0] = 'c'
TypeError: 'str' object does
not support item assignment
>>> 'c'+fruit[1:]  ⇐ you can only do this
'canana'
```

⇐ you cannot replace one of the characters! python strings are immutable!

Python string / list features are actually very powerful!
We will come back to discuss them in details at a upcoming lecture.

PUT IT ALL TOGETHER

- Testing prime numbers:

```
numbers = [17,59,83,129,187]
for m in numbers:
    for n in range(m):
        if n<2:
            continue
        if m%n == 0:
            print(m,'is a multiple of',n)
            break
    else:
        print(m,'is a prime number')
```

⇐ Surely you can do **range(2,m)** instead of the ‘continue’ statement.

```
17 is a prime number
...
187 is a multiple of 11
```

PUT IT ALL TOGETHER (II)

- Or one can get the testing number with the **raw_input** function:

```
inp = input('Please enter a number: ')
m = int(inp)
for n in range(m):
    if n<2:
        continue
    if m%n == 0:
        print (m,'is a multiple of',n)
        break
else:
    print (m,'is a prime number')
```

```
Please enter a number: 127 ⇐ input by keyboard
127 is a prime number
```

INTERMISSION

- Please try to find out:
  - Is **1237** a prime number?
    How about **12347**, **123457**, and **1234567**?
  - Print out all of the factors of **12345678**.

HANDS-ON SESSION

- Up to now we have gone through:
  - The basic structure and syntax of python
  - Variables and operators
  - Branching, conditionals, and iterations
- You should be able to write a meaningful program and carry out some interesting calculations already!
- Let's start our 2nd round of hands-on session now!

HANDS-ON SESSION

- **Practice 1:**
  Print a **multiplication table** up to 12x12 on your screen:

```
2x1 = 2
2x2 = 4
2x3 = 6
2x4 = 8
2x5 = 10
2x6 = 12
...
12x12 = 144
```

HANDS-ON SESSION

- **Practice 2:**
  Find out all of the prime numbers which are smaller than **10000**.

```
2 3 5 7 11 13 17 19 23 29 31 37 41
43 47 53 59 61 67 71 73 79 83 89
97 ... 9973
```

HANDS-ON SESSION

- **Practice 3:**
  A score and grade mapping is given below:

90–100: A+	85–89: A	80–84: A–
77–79: B+	73–76: B	70–72: B–
67–69: C+	63–66: C	60–62: C–
50–59: D	40–49: E	0–39: F

  Please write a small program which can convert the score to grade levels, e.g.:

```
Please enter a score: 95    ⇐ input by keyboard
Your grade is “A+”.  ⇐ output by your code
```