Course 5

POO

revision PROCEDURAL PROGRAMMING

Chap.5. Conditionals and Recursion
CONTENTS

- Non-arithmetic operators
  - Modulus operator
  - Boolean expressions
  - Logical operators
- Conditional execution
  - Alternative execution
  - Chained conditionals
  - Nested conditionals
- Recursion
  - Stack diagrams / flow of execution for recursive functions
  - Infinite recursion
- Keyboard input
- Debugging, Glossary, Exercises

Modulus operator

- The modulus operator works on integers and yields the remainder when the first operand is divided by the second.
- In Python, the modulus operator is a percent sign, %. The syntax is the same as for other operators:

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

- **Utility**:
  - checking whether one number is divisible by another
  - extracting the right-most digit or digits from a number

Boolean expressions

- A boolean expression is an expression that is either true or false.
- The following examples use the relational operator ==, which compares two operands and produces **True** if they are equal and **False** otherwise:

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

- **True** and **False** are special values that belong to the type bool; they are not strings.

Boolean expressions

- Other relational operators are:

x != y

x > y

x < y

x >= y

x <= y

- Look out! A common error is to use a single equal sign = instead of a double equal sign ==. Remember that = is an assignment operator and == is a relational operator.

Logical operators

- There are three logical operators: and, or, not.

x>0 and x<10

n%2 ==0 or n%3 == 0

not (x>y)

- Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as **True**.

>>> 17 and **True**

True

Conditional execution

- Conditional statements give the ability to check conditions and change the behavior of the program accordingly.
- The simplest form is the if statement:

```
if <cond>:
    <statement>
```

- e.g.

```
if x>0:
    print 'x is a positive number'
```

- There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x>0:
    pass            # need to handle negative values
```

Alternative execution

- This is the second form of the if statement.
- For an alternative execution, there are two possibilities and the condition determines which one gets executed. The alternatives are called branches because they are branches in the flow of execution.

```
if <cond>:
        <statement for true>
else:
        <statement for false>
```

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

```
if x%2:
        print <messageT>
else:
        print <messageF>
```

Chained conditionals

- Sometimes there are more than two possibilities and we need more than two branches.
- One way to express a computation like that is a chained conditional, as follows:

```
if <cond1>:
        <statement1 for true>
elif <cond2>:
        <statement2 for true>
else:
        <statement3 for false >
```

```
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’
```

```
if choice==‘a’:
        draw_a()
elif choice==‘b’:
        draw_b()
elif choice == ‘c’:
        draw_c()
```

Nested conditionals

- One conditional can also be nested within another.

```
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’
```

- Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Nested conditionals

- 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’
```

```
if 0<x and x<10:
    print ‘x is a positive single-digit number’
```

Recursion

- It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do.

```
def countdown(n):
    if n<=0:
        print 'Out of space!'
    else:
        print n
        countdown(n-1)
```

```
>>> countdown(3)
```

```
def print_n(s, n):
    if n<=0:
        return
    print s
    print_n(s,n-1)
```

- A function that calls itself is a recursive function.
- The process is called recursion.

Infinite recursion

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not a good idea.

```
def recurse():
    recurse()
```

- In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

RuntimeError: Maximum recursion depth exceeded

Keyboard input

- The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python 2 provides a built-in function called raw_input that gets input from the keyboard. In Python 3, it is called input.

- When this function is called, the program stops and waits for the user to type something. When the user presses Enter, the program resumes and raw_input / input returns what the user typed **as a string**.

```
>>> text = raw_input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

Keyboard input

- Before getting input from the user, it is a good idea to print a prompt telling the user what to input.

```
>>> name = raw_input('What is your name?\n')
What is your name?
My name is Python.
>>> name
'My name is Python.'
```

newline

- If you expect the user to type an integer, you can try to convert the return value to **int**:

```
>>> arg = 'What number do you suggest?\n'
>>> nr = raw_input(arg)
>>> int(nr)
```

Debugging, Glossary, Exercises