Conditionals and Iterations

The modulus operator

- The modulus operator works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second.
- In Python, the modulus operator is a percent sign (%).

Example

- The syntax is the same as for other operators:

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

- So 7 divided by 3 is 2 with 1 left over.

Uses

- Check whether one number is divisible by another if x % y is zero, then x is divisible by y.
- you can extract the right-most digit or digits from a number.
- For example,

  x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

Boolean expressions

- A Boolean expression is an expression that is either true or false.
- One way to write a Boolean expression is to use the operator ==, which compares two values and produces a Boolean value:

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

- True and False are special values that are built into Python.

Comparison Operators

- 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

NOTE: "= is an assignment operator and == is a comparison operator". Also, there is no such thing as =< or =>.

Logical operators

- There are three logical operators:
  - and,
  - or
  - not
- For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10.
- n%2 == 0 or n%3 == 0
- not(x > y) is true if (x > y) is false, that is, if x is less than or equal to y.

Identity operators

- Identity operators compare the memory locations of two objects. There are two Identity operators as explained below

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Bitwise Operators

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Keyboard Input

- input(): built in function to get data from keyboard.
- Takes data in the form of **string.**
- Eg:

```
>>> input1 = input ()
What are you waiting for?
>>> print input1
What are you waiting for?
```

- Before calling input, it is a good idea to print a message telling the user what to input. This message is called a **prompt.**
- A prompt can be supplied as an argument to input.

- Eg:

```
>>> name = input ("What...is your name? ")
What...is your name? Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

- If we expect the response to be an integer, then type conversion needs to be done.
- Eg:

```
prompt = "What is the airspeed velocity of an unladen swallow?"
speed =int(input(prompt))
```

Conditional Execution

- To write useful programs we need the ability to check conditions and change the behaviour of the program accordingly.
- Different conditional statements in python are:
  - IF
  - IF ---Else (Alternative Execution)
  - IF--- ELIF---- ELSE (Chained Conditionals)
  - Nested Conditionals

If Condition

- If x > 0:

  print "x is positive“

- The Boolean expression after the if statement is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens.
- Structure of If
  - HEADER:

    FIRST STATEMENT

    ...

    LAST STATEMENT

If Condition

- There is no limit on the number of statements that can appear in the body of an if statement, 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.

Alternative Execution

- A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed.
- Eg:

```
if x%2 == 0:
        print x, "is even"
else:
        print x, "is odd“
```

- The alternatives are called branches.

Continue….

- Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

Chained Conditionals

- Sometimes there are more than two possibilities and we need more than two branches.

```
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“
```

NOTE: There is no limit of the number of elif statements, but the last branch has to be an else statement

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
```

```
if 0 < x and x < 10:
    print "x is a positive single digit.“
```

- Python provides an alternative syntax that is similar to mathematical notation:

```
if 0 < x < 10:
    print "x is a positive single digit."
```

Shortcuts for Conditions

- Numeric value 0 is treated as False
- Empty sequence "", [] is treated as False
- Everything else is True

```
if m%n:
    (m,n) = (n,m%n)
else:
    gcd = n
```

Avoid Nested If

- For example, We can rewrite the following code using a single conditional:

```
if 0 < x:
        if x < 10:
            print "x is a positive single digit.“
```

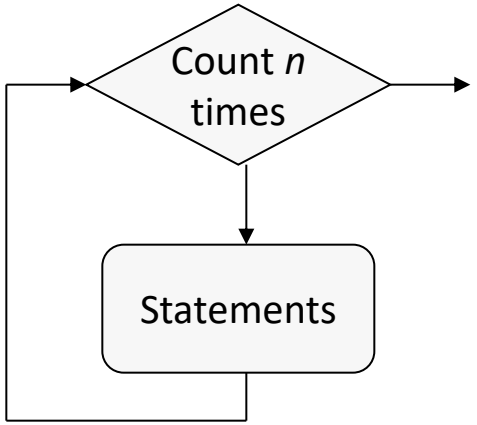
Better way:

```
if 0 < x and x < 10:
        print "x is a positive single digit."
```

ITERATION

Doing the Same Thing Many Times

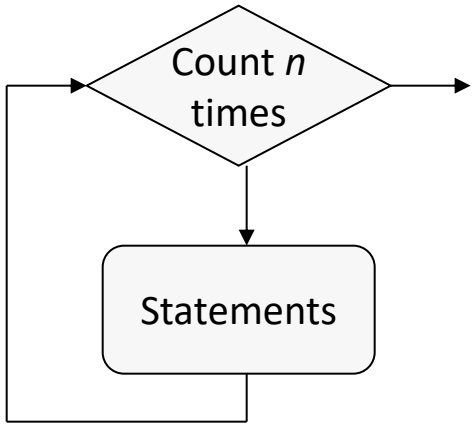
- It's possible to do something repeatedly by just writing it all out
- Print 'hello' 5 times




```
>>> print('Hello!')
Hello
>>> print('Hello!')
Hello
>>> print('Hello!')
Hello
>>> print('Hello!')
Hello
>>> print('Hello!')
Hello
```

Iteration and Loops

- A *loop* repeats a sequence of statements
- A *definite loop* repeats a sequence of statements a predictable number of times




```
>>> for x in range(5): print('Hello!')
...
Hello
Hello
Hello
Hello
Hello
```

The for Loop

- Python's **for** loop can be used to iterate a definite number of times

for <*variable*> in range(<*number of times*>): <*statement*>

- Use this syntax when you have only one statement to repeat

```
for <variable> in range(<number of times>):
    <statement-1>
    <statement-2>
    …
    <statement-n>
```

- Use *indentation* to format two or more statements below the *loop header*

```
>>> for x in range(3):
...     print('Hello!')
...     print('goodbye')
...
Hello!
goodbye
Hello!
goodbye
Hello!
goodbye
```

Using the Loop Variable

- The *loop variable* picks up the next value in a *sequence* on each pass through the loop
- The expression **`range(n)`** generates a sequence of **`int`**s from **`0`** through **`n - 1`**

loop variable

```
>>> for x in range(5): print(x)
...
0
1
2
3
4
>>> list(range(5)) # Show as a list
[0, 1, 2, 3, 4]
```

Counting from 1 through *n*

- The expression **range(low, high)** generates a sequence of **int**s from **low** through **high - 1**

```
>>> for x in range(1, 6): print(x)
...
1
2
3
4
5
```

Counting from n through *1*

- The expression **range(high, low, step)** generates a sequence of **int**s from **high** through **low+1.**

```
>>> for x in range(6, 1, -1): print(x)
...
6
5
4
3
2
```

Skipping Steps in a Sequence

- The expression **range(low, high, step)** generates a sequence of **int**s starting with **low** and counting by **step** until **high - 1** is reached or exceeded

```
>>> for x in range(1, 6, 2): print(x)
...
1
3
5
>>> list(range(1, 6, 2)) # Show as a list
[1, 3, 5]
```

While Loop

A for loop is used when a program knows it needs to repeat a block of code for a certain number of times.

A while loop is used when a program needs to loop until a particular condition occurs.

```
i = 0
while i < 10:
    print(i)
    i += 1
```

Flow of Execution for WHILE Statement

1. Evaluate the condition, yielding 0 or 1.
2. If the condition is false (0), exit the `while` statement and continue execution at the next statement.
3. If the condition is true (1), execute each of the statements in the body and then go back to step 1.

Looping Until User Wants To Quit

```
quit = "n"
while quit == "n":
    quit = input("Do you want to quit? ")
```

Range Function

- Range returns an immutable sequence objects of integers between the given start integer to the stop integer.
- range() constructor has two forms of definition:
  - range(stop)
  - range(start, stop, step)
    - **start** - integer starting from which the sequence of integers is to be returned
    - integer before which the sequence of integers is to be returned.
      The range of integers end at **stop - 1**.
      - **step (Optional)** - integer value which determines the increment between each integer in the sequence