If Statement

Most computations require different actions for different inputs and one way to express these differences in Python is using the if statement.

```python
if <boolean expression>:
    <statement>
    <statement>
    ...
elif <boolean expression>:
    <statement>
    <statement>
    ...
elif <boolean expression>:
    <statement>
    <statement>
    ...
else:
    <statement>
    <statement>
    ...
```

Flip.py: Simulate a coin flip by writing 'Heads' or 'Tails' to standard output.

```python
import random
import stdio
if random.randrange(0, 2) == 0:
    stdio.writeln('Heads')
else:
    stdio.writeln('Tails')
```

$ python3 flip.py
Tails
$ python3 flip.py
Heads
$ python3 flip.py
Heads
$ python3 flip.py
Tails
$ python3 flip.py
Heads
While Statement

Many computations are inherently repetitive and the basic Python construct for handling such computations is the while statement

```python
while <boolean expression>:
    <statement>
    <statement>
    ...
```

5 / 18

While Statement

```python
tenhellos.py: Write 10 Hellos to standard output.
import stdio
stdio.writeln('1st Hello')
stdio.writeln('2nd Hello')
stdio.writeln('3rd Hello')
i = 4
while i <= 10:
    stdio.writeln(str(i) + 'th Hello')
    i = i + 1
```

Variable trace

<table>
<thead>
<tr>
<th>i</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4th hello</td>
</tr>
<tr>
<td>5</td>
<td>5th hello</td>
</tr>
<tr>
<td>6</td>
<td>6th hello</td>
</tr>
<tr>
<td>7</td>
<td>7th hello</td>
</tr>
<tr>
<td>8</td>
<td>8th hello</td>
</tr>
<tr>
<td>9</td>
<td>9th hello</td>
</tr>
<tr>
<td>10</td>
<td>10th hello</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

6 / 18

While Statement

```
powersoftwo.py: Accept positive integer n as a command-line argument. Write to standard output a table showing the first n powers of two.
import stdio
import sys
n = int(sys.argv[1])
power = 1
i = 0
while i <= n:
    stdio.writeln(str(i) + ' ' + str(power))
    power = 2 * power
    i = i + 1
```

Variable trace (n = 8)

<table>
<thead>
<tr>
<th>power</th>
<th>i</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4 16</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5 32</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6 64</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7 128</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8 256</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

7 / 18

While Statement

```
Modifying a variable is something that we do so often that Python provides shorthand notations for the purpose

The most common practice is to abbreviate an assignment statement of the form

```python
i = i + 1
```

with the shorthand notation

```python
i += 1
```

The same notation works for other binary operators, including -, *, and /

The scope of a variable is part of the program where it is defined, ie, statements that follow the definition in the same block (marked by indentation level)
```
For Statement

The for statement provides a more compact notation for carrying out repeated computations

```
for <variable> in <iterable object>:
    <statement>
    ...
```

The most commonly used iterable objects are the lists containing arithmetic progressions of integers, returned by the built-in function `range()`. The call `range(start, stop[, step])` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) given by the optional `step` argument, which defaults to 1.

The call `range(stop)` is shorthand for `range(0, stop)`

For example

```
range(8, 0, -2) = [8, 6, 4, 2]
range(3, 9) = [3, 4, 5, 6, 7, 8]
range(5) = [0, 1, 2, 3, 4]
```

For Statement

The `tenhellos.py` program can be written using a for statement as follows

```
import stdio
stdio.writeln('1st Hello ')
stdio.writeln('2nd Hello ')
stdio.writeln('3rd Hello ')
for i in range(4, 11):
    stdio.writeln(str(i) + 'th Hello ')
```

Strings are iterable objects, so its characters can be enumerated using a for loop

For example, the following code

```
import stdio
for c in 'AGCT ':
    stdio.writeln(c)
```

produces the output

```
A
G
C
T
```

Nesting

The if, while, and for statements, collectively called control-flow statements, have the same status as assignment statements or any other statements in Python.

As a result, we can use a control-flow statement whenever a statement is called for.

In particular, we can nest one or more of the control-flow statements in the body of another

```
divisorpattern.py: Accept integer command-line argument n. Write to standard output an n-by-n table with an asterisk in row i and column j if either i divides j or j divides i.
```

```
import stdio
import sys
n = int(sys.argv[1])
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if (i % j == 0) or (j % i == 0):
            stdio.write('* ')
        else:
            stdio.write(' ')
    stdio.writeln(i)
```

```
$ python3 divisorpattern.py 3
* * * 1
  * 2
  * 3
```

Variable trace (n = 3)

```
i j output
----------------
1 1 'x'
1 2 'x ' 1
1 3 'x 1
2 1 'x '
2 2 'x ' 2
2 3 'x 2
3 1 'x '
3 2 'x ' 3
3 3 'x 3
```

```
$ python3 divisorpattern.py 10
* * * * * * * * * * 1
  * * * * * * * * * 2
  * * * * * * * * 3
  * * * * * * * 4
  * * * * * * 5
  * * * * * 6
  * * * * * 7
  * * * * 8
  * * * 9
  * * 10
```

```
$ python3 divisorpattern.py 10
* * * * * * * * * * 1
  * * * * * * * * * 2
  * * * * * * * * 3
  * * * * * * * 4
  * * * * * * 5
  * * * * * 6
  * * * * * 7
  * * * * 8
  * * * * 9
  * * * 10
```
Applications

harmonic.py: Accept integer \( n \) as a command-line argument. Write to standard output the \( n \)th harmonic number \( H_n \), computed as \( H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \). Note that \( H_n \approx \ln(n) + 0.57721 \) for large \( n \).

```python
import stdio
import sys
n = int(sys.argv[1])
total = 0.0
for i in range(1, n + 1):
total += 1 / i
stdio.writeln(total)
```

$ python3 harmonic.py 2
1.5
$ python3 harmonic.py 10
2.92896825397
$ python3 harmonic.py 10000
9.78760603604

Applications

sqrt.py: Accept a float \( c \) as a command-line argument. Write to standard output the square root of \( c \) to 15 decimal places of accuracy, calculated using Newton's method.

```python
import stdio
import sys
EPSILON = 1e-15
c = float(sys.argv[1])
t = c
while abs(1 - c / t ** 2) > EPSILON:
t = (c / t + t) / 2
stdio.writeln(t)
```

$ python3 sqrt.py 2.0
1.41421356237
$ python3 sqrt.py 2544545
1595.16300108

Applications

binary.py: Accept integer \( n \) as a command-line argument. Write the binary representation of \( n \) to standard output.

```python
import sys
import stdio
n = int(sys.argv[1])
v = 1
while v <= n // 2:
v *= 2
while v > 0:
if n < v:
stdio.write(0)
else:
stdio.write(1)
n -= v
v //= 2
stdio.writeln()
```

$ python3 binary.py 19
10011
$ python3 binary.py 255
1111111
$ python3 binary.py 512
100000000
$ python3 binary.py 1000000000
1101110011010110011000100000000

Applications

gambler.py: Accept integer command-line arguments \( stake \), \( goal \), and \( trials \). Run \( trials \) experiments that start with \( stake \) dollars and terminate on 0 dollars or \( goal \). Write to standard output the percentage of wins and the average number of bets per experiment, which can be calculated as \( 100 \times \frac{stake}{goal} \) and \( \frac{stake}{goal - stake} \), respectively.

```python
import random
import stdio
import sys
stake = int(sys.argv[1])
goal = int(sys.argv[2])
trials = int(sys.argv[3])
bets = 0
wins = 0
for t in range(trials):
cash = stake
while cash > 0 and cash < goal:
bets += 1
if random.randrange(0, 2) == 0:
cash += 1
else:
cash -= 1
if cash == goal:
wins += 1
stdio.writeln(100 * wins // trials + '% wins ')
stdio.writeln('Avg # bets : ' + str(bets // trials))
```

$ python3 gambler.py 50 250 100
24% wins
Avg # bets : 12548
$ python3 gambler.py 500 2500 100
27% wins
Avg # bets : 1124855
import stdio
import sys
n = int(sys.argv[1])
factor = 2
while factor * factor <= n:
    while n % factor == 0:
        n //= factor
        stdio.write(str(factor) + ' ')
    factor += 1
if n > 1:
    stdio.write(n)
stdio.writeln()

$ python3 factors.py 3757208
2 2 2 7 13 13 397
$ python3 factors.py 287994837222311
17 1739347 9739789

Other Conditional and Loop Constructs

The conditional expression supports an alternate form of an if-else statement

<expression> if <boolean expression> else <expression>

For example, the following code assigns 'Heads' or 'Tails' to the variable flip, each with probability 1/2

flip = 'Heads' if random.random() < 0.5 else 'Tails'

The break statement immediately exits a loop without letting it to run to completion

For example, the following code tests and prints if a number \( N \) is prime or not

i = 2
while i <= N // i:
    if N % i == 0:
        break
    i += 1
stdio.writeln(str(N) + (' is ' if i > N // i else ' is not ') + 'prime')

The continue statement skips to next iteration of a loop

For example, the following code iterates over the characters in the English alphabet and prints only the consonants

for c in 'abcdeghijklmnopqrstuvwxyz':
    if c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u':
        continue
    stdio.writeln(c)