Outline

1. If Statement
2. While Statement
3. For Statement
4. Nesting
5. Applications
6. Other Conditional and Loop Constructs
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>
    ...
```
If 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
Many computations are inherently repetitive and the basic Python construct for handling such computations is the `while` statement

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

tenhellos.py: Write 10 Hellos to standard output.

```python
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

$ python3 tenhellos.py
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

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>
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.

```python
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
```

```
$ python3 powersoftwo.py 8
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
```

Variable trace (\( n = 8 \))

<table>
<thead>
<tr>
<th>power</th>
<th>i</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0 1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1 2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2 4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3 8</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4 16</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>5 32</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>6 64</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>7 128</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>8 256</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
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

\[ i = i + 1 \]

with the shorthand notation

\[ i += 1 \]

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

The scope of a variable is part of the program where it is defined, i.e., 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>
    <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:

```python
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:

```python
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.
Nesting

divisorditern.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 \( i \).
Applications

harmonic.py: Accept integer n as a command-line argument. Write to standard output the nth harmonic number $H_n$, computed as $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$. 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
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
11111111
$ python3 binary.py 512
1000000000
$ python3 binary.py 100000000
110111001101011001010000000
Applications

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

```python
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

\[
\text{<expression>} \text{ if } \text{<boolean expression>} \text{ else } \text{<expression>}
\]

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

\[
\text{flip} = \text{'Heads'} \text{ if random.random()} < 0.5 \text{ else 'Tails'}
\]

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

\[
i = 2
\text{while } i \leq N // i:
\quad \text{if } N \% i == 0:
\quad \quad \text{break}
\quad i += 1
\text{stdio.writeln(str(N) + (' is ' if i > N // i else ' is not ') + 'prime')}
\]

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

\[
\text{for } c \text{ in 'abcdefgijklmnopqrstuvwxyz':}
\quad \text{if } c == 'a' \text{ or } c == 'e' \text{ or } c == 'i' \text{ or } c == 'o' \text{ or } c == 'u':
\quad \quad \text{continue}
\quad \text{stdio.writeln(c)}
\]