Control Flow

Outline

1. Branching
2. Looping
3. Nesting
4. Scope of Variables
5. Applications

Branching

Branching

If statement

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

Branching

Branching

Program: `grade.py`

Branching

Program: `grade.py`

- Command-line input: a percentage *score* (float)

Branching

Program: `grade.py`

- Command-line input: a percentage *score* (float)
- Standard output: the corresponding letter grade

Branching

Program: `grade.py`

- Command-line input: a percentage *score* (float)
- Standard output: the corresponding letter grade

```
>_ ~/workspace/ipp/programs

$ _
```

Branching

Program: `grade.py`

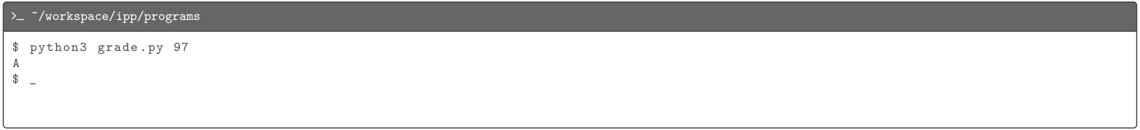
- Command-line input: a percentage *score* (float)
- Standard output: the corresponding letter grade

```
>_ ~/workspace/ipp/programs

$ python3 grade.py 97
```

Branching

Program: `grade.py`

- Command-line input: a percentage *score* (float)
- Standard output: the corresponding letter grade



```
$ python3 grade.py 97
A
$ _
```

Branching

Program: `grade.py`

- Command-line input: a percentage *score* (float)
- Standard output: the corresponding letter grade

```
>_ ~/workspace/ipp/programs
```

```
$ python3 grade.py 97
A
$ python3 grade.py 56
```

Branching

Program: `grade.py`

- Command-line input: a percentage *score* (float)
- Standard output: the corresponding letter grade

```
>_ ~/workspace/ipp/programs
```

```
$ python3 grade.py 97
A
$ python3 grade.py 56
F
$ _
```

Branching

Branching

grade.py

```
import stdio
import sys

score = float(sys.argv[1])
if score >= 93:
    stdio.writeln("A")
elif score >= 90:
    stdio.writeln("A-")
elif score >= 87:
    stdio.writeln("B+")
elif score >= 83:
    stdio.writeln("B")
elif score >= 80:
    stdio.writeln("B-")
elif score >= 77:
    stdio.writeln("C+")
elif score >= 73:
    stdio.writeln("C")
elif score >= 70:
    stdio.writeln("C-")
elif score >= 67:
    stdio.writeln("D+")
elif score >= 63:
    stdio.writeln("D")
elif score >= 60:
    stdio.writeln("D-")
else:
    stdio.writeln("F")
```

Branching

Branching

Conditional expression

```
... <expression1> if <expression> else <expression2> ...
```

Branching

Branching

Program: `flip.py`

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”

```
>_ ~/workspace/ipp/programs

$ _
```

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”

```
>_ ~/workspace/ipp/programs

$ python3 flip.py
```

Branching

Program: `flip.py`

- Standard output: "heads" or "tails"

```
>_ ~/workspace/ipp/programs

$ python3 flip.py
Heads
$ _
```

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”

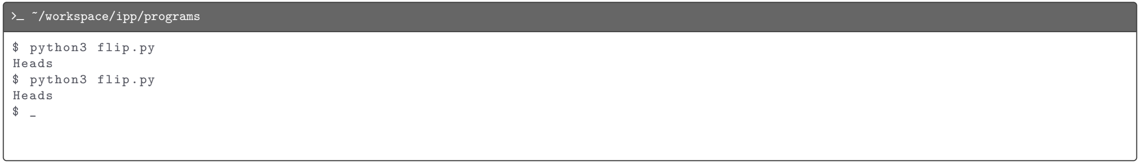
>_ ~/workspace/ipp/programs

```
$ python3 flip.py
Heads
$ python3 flip.py
```

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”

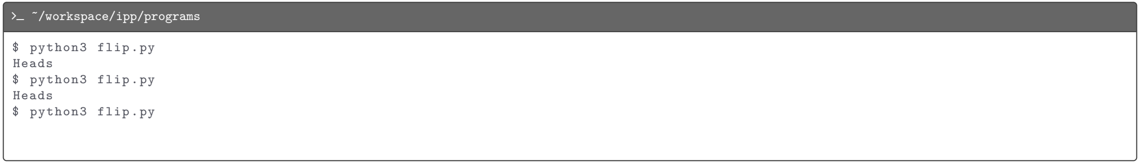


```
$ python3 flip.py
Heads
$ python3 flip.py
Heads
$ _
```

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”



```
>_ ~/workspace/ipp/programs
```

```
$ python3 flip.py
Heads
$ python3 flip.py
Heads
$ python3 flip.py
```

Branching

Program: `flip.py`

- Standard output: “heads” or “tails”

```
>_ ~/workspace/ipp/programs
```

```
$ python3 flip.py
Heads
$ python3 flip.py
Heads
$ python3 flip.py
Tails
$ _
```

Branching

Branching

flip.py

```python
import stdio
import stdrandom

result = "Heads" if stdrandom.bernoulli() else "Tails"
stdio.writeln(result)
```

Looping

Looping

While statement

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

Looping

Looping

Program: `nhellos.py`

Looping

Program: `nhellos.py`

- Command-line input: n (int)

Looping

Program: `nhellos.py`

- Command-line input: n (int)
- Standard output: n Hellos

Looping

Program: `nhellos.py`

- Command-line input: *n* (int)
- Standard output: *n* Hellos

```
>_ ~/workspace/ipp/programs

$ _
```

Looping

Program: `nhellos.py`

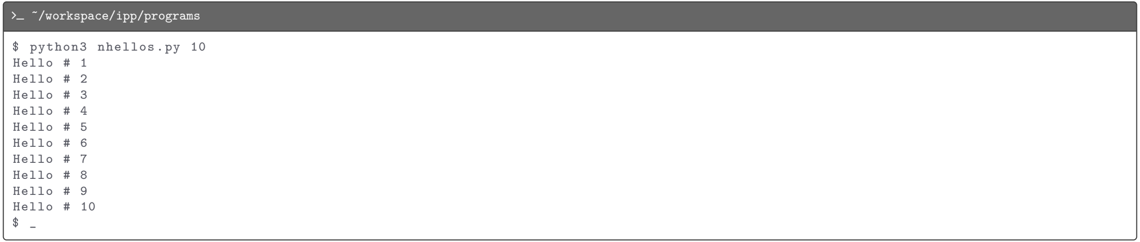
- Command-line input: *n* (int)
- Standard output: *n* Hellos

```
>_ ~/workspace/ipp/programs
$ python3 nhellos.py 10
```

Looping

Program: `nhellos.py`

- Command-line input: *n* (int)
- Standard output: *n* Hellos



```
>_ ~/workspace/ipp/programs
```

```
$ python3 nhellos.py 10
Hello # 1
Hello # 2
Hello # 3
Hello # 4
Hello # 5
Hello # 6
Hello # 7
Hello # 8
Hello # 9
Hello # 10
$ _
```

Looping

Looping

nhellos.py

```python
import stdio
import sys

n = int(sys.argv[1])
i = 1
while i <= n:
    stdio.writeln("Hello # " + str(i))
    i += 1
```

Looping

Looping

Variable trace ($n = 3$)

nhellos.py

```
1  import stdio
2  import sys
3
4  n = int(sys.argv[1])
5  i = 1
6  while i <= n:
7      stdio.writeln("Hello # " + str(i))
8      i += 1
```

line #	n	i
4	3	
5	3	1
6	3	1
7	3	1
8	3	2
6	3	2
7	3	2
8	3	3
6	3	3
7	3	3
8	3	4
6	3	4

Looping

Looping

For statement

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

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

The call `range(start, stop)` is shorthand for `range(start, stop, 1)`

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

The call `range(start, stop)` is shorthand for `range(start, stop, 1)`

The call `range(stop)` is shorthand for `range(0, stop, 1)`

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

The call `range(start, stop)` is shorthand for `range(start, stop, 1)`

The call `range(stop)` is shorthand for `range(0, stop, 1)`

Example:

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

The call `range(start, stop)` is shorthand for `range(start, stop, 1)`

The call `range(stop)` is shorthand for `range(0, stop, 1)`

Example:

- `range(8, 0, -2)` returns `[8, 6, 4, 2]`

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

The call `range(start, stop)` is shorthand for `range(start, stop, 1)`

The call `range(stop)` is shorthand for `range(0, stop, 1)`

Example:

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

Looping

For statement

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

Most commonly used iterable objects are lists containing arithmetic progressions of integers

The built-in function call `range(start, stop, step)` returns a list starting at `start`, ending just before `stop`, and in increments (or decrements) of `step`

The call `range(start, stop)` is shorthand for `range(start, stop, 1)`

The call `range(stop)` is shorthand for `range(0, stop, 1)`

Example:

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

Looping

Looping

Program: `powersoftwo.py`

Looping

Program: `powersoftwo.py`

- Command-line input: n (int)

Looping

Program: `powersoftwo.py`

- Command-line input: n (int)
- Standard output: a table of powers of 2 that are less than or equal to 2^n

Looping

Program: `powersoftwo.py`

- Command-line input: n (int)
- Standard output: a table of powers of 2 that are less than or equal to 2^n

```
>_ ~/workspace/ipp/programs

$ _
```

Looping

Program: `powersoftwo.py`

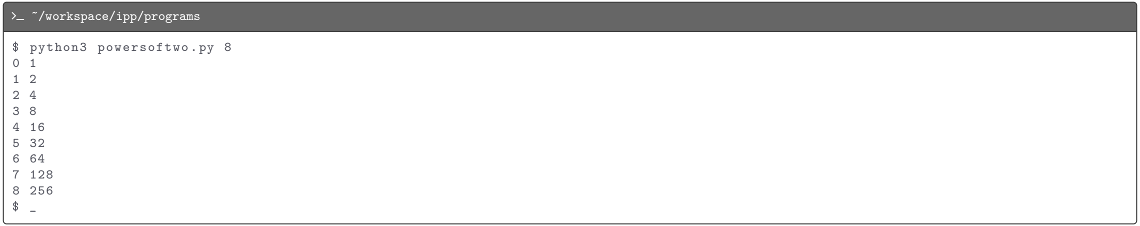
- Command-line input: n (int)
- Standard output: a table of powers of 2 that are less than or equal to 2^n

```
>_ ~/workspace/ipp/programs

$ python3 powersoftwo.py 8
```

Looping

Program: `powersoftwo.py`

- Command-line input: n (int)
- Standard output: a table of powers of 2 that are less than or equal to 2^n



```
>_ ~/workspace/ipp/programs
```

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

Looping

Looping

powersoftwo.py

```python
import stdio
import sys

n = int(sys.argv[1])
power = 1
for i in range(n + 1):
    stdio.writeln(str(i) + " " + str(power))
    power *= 2
```

Looping

Looping

Variable trace ($n = 3$)

powersoftwo.py

```
1  import stdio
2  import sys
3
4  n = int(sys.argv[1])
5  power = 1
6  for i in range(n + 1):
7      stdio.writeln(str(i) + " " + str(power))
8      power *= 2
```

line #	n	power	i
4	3		
5	3	1	
6	3	1	0
7	3	1	0
8	3	2	0
6	3	2	1
7	3	2	1
8	3	4	1
6	3	4	2
7	3	4	2
8	3	8	2
6	3	8	3
7	3	8	3
8	3	16	3

Looping

Looping

Strings are iterable objects — its characters can be enumerated using a for statement

Looping

Strings are iterable objects — its characters can be enumerated using a for statement

Example

```
import stdio

for c in "Python’s great!":
    stdio.write(c + " ")
stdio.writeln()
```

Looping

Strings are iterable objects — its characters can be enumerated using a for statement

Example

```
import stdio

for c in "Python’s great!":
    stdio.write(c + " ")
stdio.writeln()
```

```
P y t h o n ’ s   g r e a t !
```

Looping

Looping

Break statement

```
break
```

Looping

Break statement

```
break
```

Example

```
n = 10
i = 0
while True:
    if i == n:
        break
    stdio.write(str(i) + " ")
    i += 2
stdio.writeln()
```

Looping

Break statement

```
break
```

Example

```
n = 10
i = 0
while True:
    if i == n:
        break
    stdio.write(str(i) + " ")
    i += 2
stdio.writeln()
```

```
0 2 4 6 8
```

Looping

Looping

Continue statement

```
continue
```

Looping

Continue statement

```
continue
```

Example

```
for i in range(10):
    if i % 2 == 0:
        continue
    stdio.write(str(i) + " ")
stdio.writeln()
```

Looping

Continue statement

```
continue
```

Example

```
for i in range(10):
    if i % 2 == 0:
        continue
    stdio.write(str(i) + " ")
stdio.writeln()
```

```
1 3 5 7 9
```

Nesting

Nesting

The if, while, and for statements can be nested within one another

Nesting

Nesting

Program: `divisorpattern.py`

Nesting

Program: `divisorpattern.py`

- Command-line input: n (int)

Nesting

Program: `divisorpattern.py`

- Command-line input: n (int)
- Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise

Nesting

Program: `divisorpattern.py`

- Command-line input: *n* (int)
- Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise

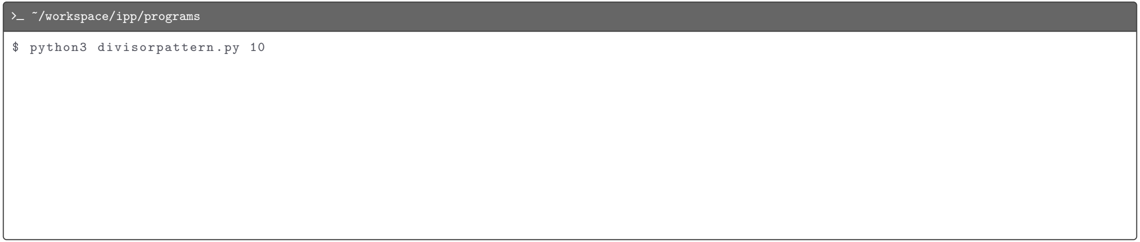
```
>_ ~/workspace/ipp/programs
```

```
$ _
```

Nesting

Program: `divisorpattern.py`

- Command-line input: n (int)
- Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise



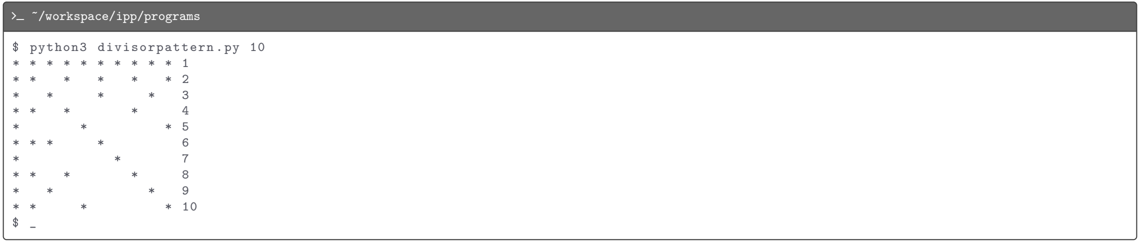
```
>_ ~/workspace/ipp/programs
```

```
$ python3 divisorpattern.py 10
```

Nesting

Program: `divisorpattern.py`

- Command-line input: n (int)
- Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise



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

Nesting

Nesting

divisorpattern.py

```python
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)
```

Nesting

Nesting

Variable trace ($n = 3$)

divisorpattern.py

```
1  import stdio
2  import sys
3
4  n = int(sys.argv[1])
5  for i in range(1, n + 1):
6      for j in range(1, n + 1):
7          if i % j == 0 or j % i == 0:
8              stdio.write("* ")
9          else:
10             stdio.write("  ")
11     stdio.writeln(i)
```

line #	n	i	j
4	3		
5	3	1	
6	3	1	1
7	3	1	1
8	3	1	1
6	3	1	2
7	3	1	2
8	3	1	2
6	3	1	3
7	3	1	3
8	3	1	3
11	3	1	
5	3	2	
6	3	2	1
7	3	2	1
8	3	2	1
6	3	2	2
7	3	2	2
8	3	2	2
6	3	2	3
7	3	2	3
10	3	2	3
11	3	2	
5	3	3	
6	3	3	1
7	3	3	1
8	3	3	1
6	3	3	2
7	3	3	2
10	3	3	2
6	3	3	3
7	3	3	3
8	3	3	3
11	3	3	

Scope of Variables

Scope of Variables

The scope of a variable is the part of the program that can refer to that variable by name

Scope of Variables

The scope of a variable is the part of the program that can refer to that variable by name

Example

divisorpattern.py

```
 1  import stdio
 2  import sys
 3
 4  n = int(sys.argv[1])
 5  for i in range(1, n + 1):
 6      for j in range(1, n + 1):
 7          if i % j == 0 or j % i == 0:
 8              stdio.write("* ")
 9          else:
10              stdio.write("  ")
11      stdio.writeln(i)
```

Variable	Scope
n	lines 4 — 11
i	lines 5 — 11
j	lines 6 — 10

Applications

Applications

Program: `harmonic.py`

Applications

Program: `harmonic.py`

- Command-line input: n (int)

Applications

Program: `harmonic.py`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

Applications

Program: `harmonic.py`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/ipp/programs

$ _
```

Applications

Program: `harmonic.py`

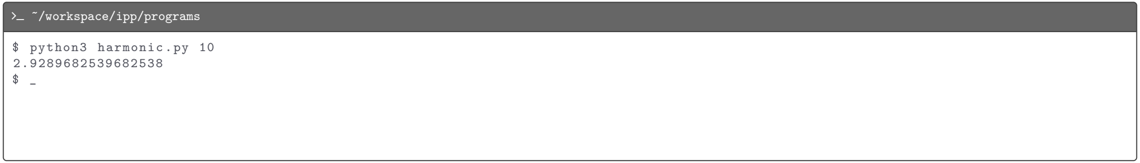
- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/ipp/programs

$ python3 harmonic.py 10
```

Applications

Program: `harmonic.py`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$



```
>_ ~/workspace/ipp/programs
```

```
$ python3 harmonic.py 10
2.9289682539682538
$ _
```

Applications

Program: `harmonic.py`

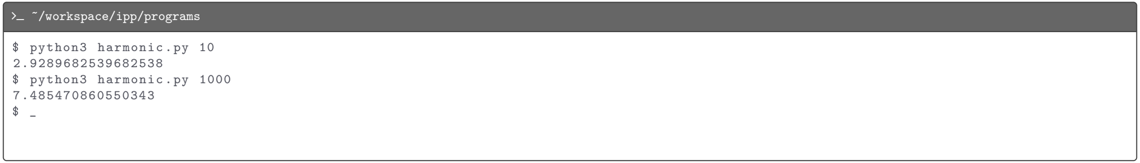
- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/ipp/programs

$ python3 harmonic.py 10
2.9289682539682538
$ python3 harmonic.py 1000
```

Applications

Program: `harmonic.py`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$



```
$ python3 harmonic.py 10
2.9289682539682538
$ python3 harmonic.py 1000
7.485470860550343
$ _
```

Applications

Program: `harmonic.py`

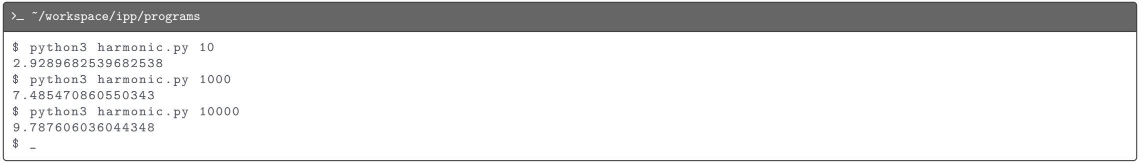
- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/ipp/programs

$ python3 harmonic.py 10
2.9289682539682538
$ python3 harmonic.py 1000
7.485470860550343
$ python3 harmonic.py 10000
```

Applications

Program: `harmonic.py`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$



```
$ python3 harmonic.py 10
2.9289682539682538
$ python3 harmonic.py 1000
7.485470860550343
$ python3 harmonic.py 10000
9.787606036044348
$ _
```

Applications

Applications

harmonic.py

```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)
```

Applications

Applications

Program: `sqrt.py`

Applications

Program: `sqrt.py`

- Command-line input: c (float)

Applications

Program: `sqrt.py`

- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places

Applications

Program: `sqrt.py`

- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places

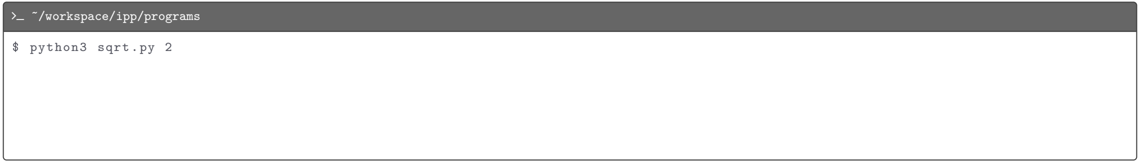
```
>_ ~/workspace/ipp/programs
```

```
$ _
```

Applications

Program: `sqrt.py`

- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places



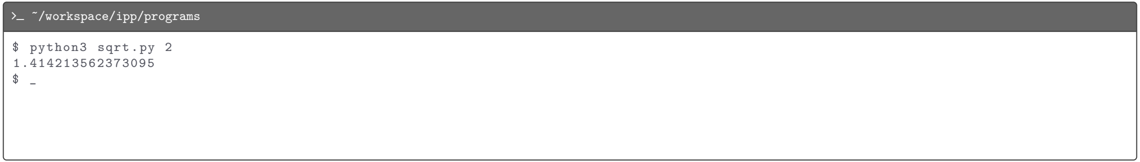
```
>_ ~/workspace/ipp/programs
```

```
$ python3 sqrt.py 2
```

Applications

Program: `sqrt.py`

- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places



```
>_ ~/workspace/ipp/programs
```

```
$ python3 sqrt.py 2
1.414213562373095
$ _
```

Applications

Program: `sqrt.py`

- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places

```
>_ ~/workspace/ipp/programs

$ python3 sqrt.py 2
1.414213562373095
$ python3 sqrt.py 1000000
```

Applications

Program: `sqrt.py`

- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places

```
>_ ~/workspace/ipp/programs

$ python3 sqrt.py 2
1.414213562373095
$ python3 sqrt.py 1000000
1000.0
$ _
```

Applications

Program: `sqrt.py`

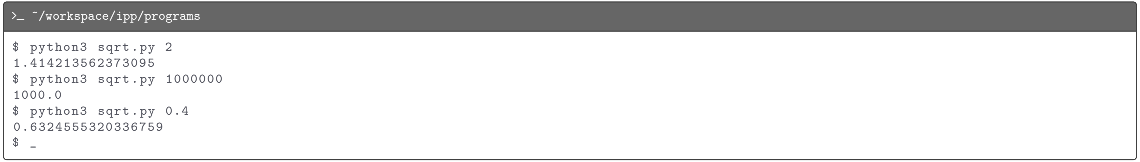
- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places

```
>_ ~/workspace/ipp/programs

$ python3 sqrt.py 2
1.414213562373095
$ python3 sqrt.py 1000000
1000.0
$ python3 sqrt.py 0.4
```

Applications

Program: `sqrt.py`

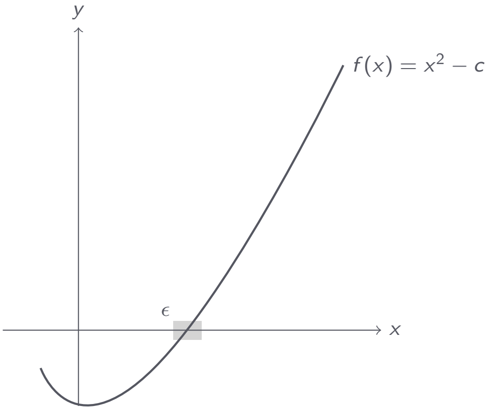
- Command-line input: c (float)
- Standard output: $\sqrt{c}$ up to 15 decimal places



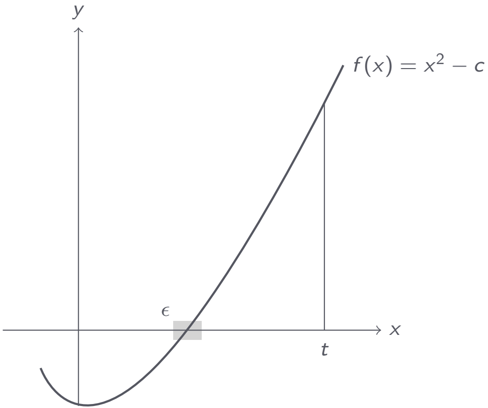
```
>_ ~/workspace/ipp/programs

$ python3 sqrt.py 2
1.414213562373095
$ python3 sqrt.py 1000000
1000.0
$ python3 sqrt.py 0.4
0.6324555320336759
$ _
```

Applications

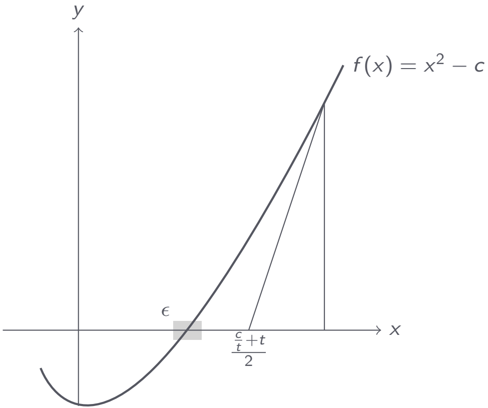
Applications



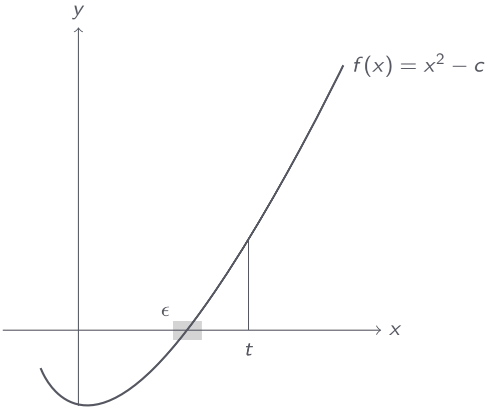
Applications



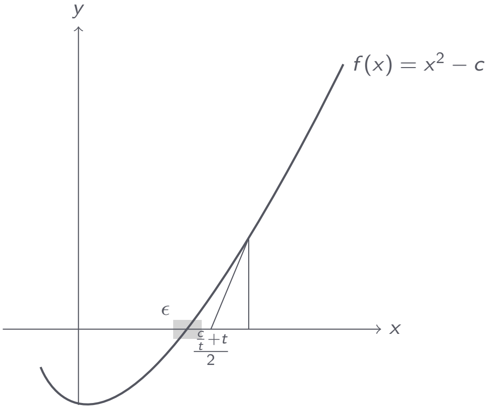
Applications



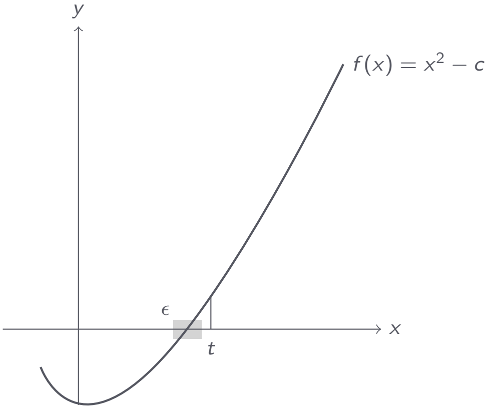
Applications



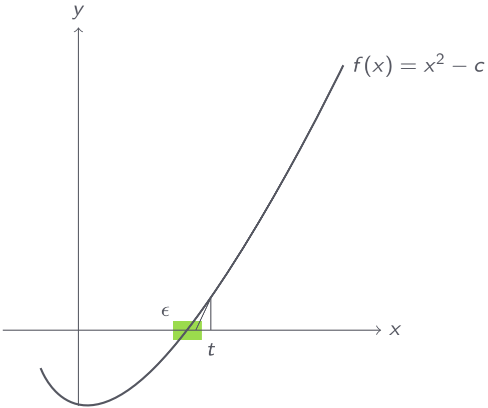
Applications



Applications



Applications



Applications

Applications

sqrt.py

```python
import stdio
import sys

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

Applications

Applications

Program: `binary.py`

Applications

Program: `binary.py`

- Command-line input: n (int)

Applications

Program: `binary.py`

- Command-line input: n (int)
- Standard output: binary representation of n

Applications

Program: `binary.py`

- Command-line input: *n* (int)
- Standard output: binary representation of *n*

```
>_ ~/workspace/ipp/programs

$ _
```

Applications

Program: `binary.py`

- Command-line input: *n* (int)
- Standard output: binary representation of *n*

```
>_ ~/workspace/ipp/programs

$ python3 binary.py 19
```

Applications

Program: `binary.py`

- Command-line input: *n* (int)
- Standard output: binary representation of *n*

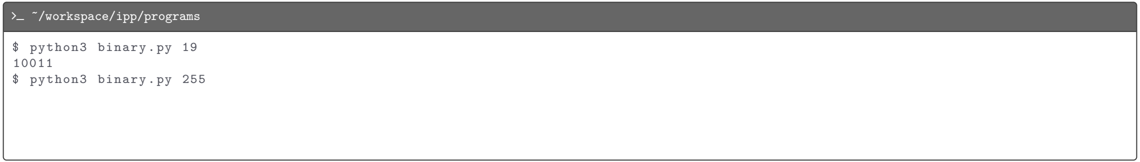
```
>_ ~/workspace/ipp/programs
```

```
$ python3 binary.py 19
10011
$ _
```

Applications

Program: `binary.py`

- Command-line input: *n* (int)
- Standard output: binary representation of *n*



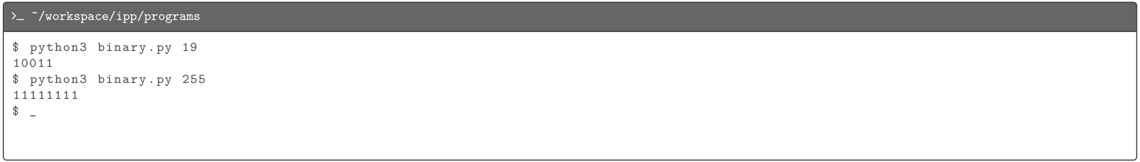
```
>_ ~/workspace/ipp/programs
```

```
$ python3 binary.py 19
10011
$ python3 binary.py 255
```

Applications

Program: `binary.py`

- Command-line input: *n* (int)
- Standard output: binary representation of *n*

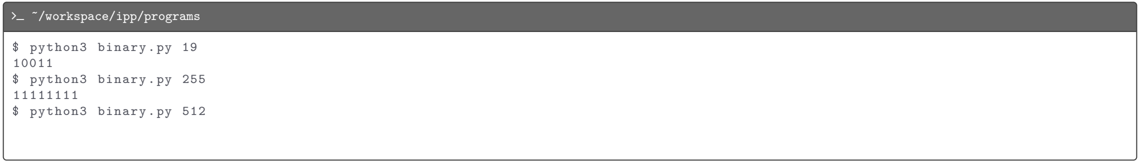


```
$ python3 binary.py 19
10011
$ python3 binary.py 255
11111111
$ _
```

Applications

Program: `binary.py`

- Command-line input: *n* (int)
- Standard output: binary representation of *n*



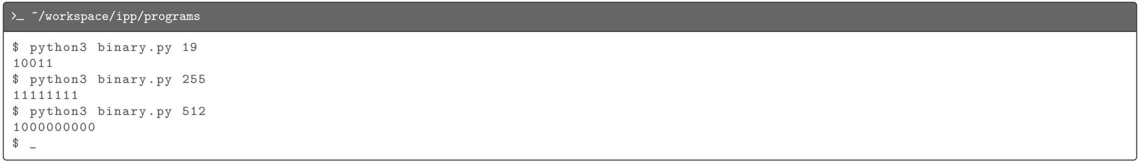
```
>_ ~/workspace/ipp/programs
```

```
$ python3 binary.py 19
10011
$ python3 binary.py 255
11111111
$ python3 binary.py 512
```

Applications

Program: `binary.py`

- Command-line input: n (int)
- Standard output: binary representation of n



```
>_ ~/workspace/ipp/programs
```

```
$ python3 binary.py 19
10011
$ python3 binary.py 255
11111111
$ python3 binary.py 512
1000000000
$ _
```

Applications

Applications

binary.py

```python
import stdio
import sys

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()
```

Applications

Applications

Program: `gambler.py`

Applications

Program: `gambler.py`

- Command-line input: *stake* (int), *goal* (int), and *trials* (int)

Applications

Program: `gambler.py`

- Command-line input: *stake* (int), *goal* (int), and *trials* (int)
- Standard output: percentage of wins and average number of bets per experiment

Applications

Program: `gambler.py`

- Command-line input: *stake* (int), *goal* (int), and *trials* (int)
- Standard output: percentage of wins and average number of bets per experiment

```
>_ ~/workspace/ipp/programs
```

```
$ _
```

Applications

Program: `gambler.py`

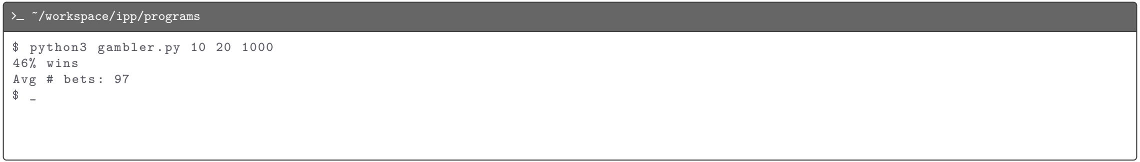
- Command-line input: *stake* (int), *goal* (int), and *trials* (int)
- Standard output: percentage of wins and average number of bets per experiment

```
>_ ~/workspace/ipp/programs

$ python3 gambler.py 10 20 1000
```

Applications

Program: `gambler.py`

- Command-line input: *stake* (int), *goal* (int), and *trials* (int)
- Standard output: percentage of wins and average number of bets per experiment



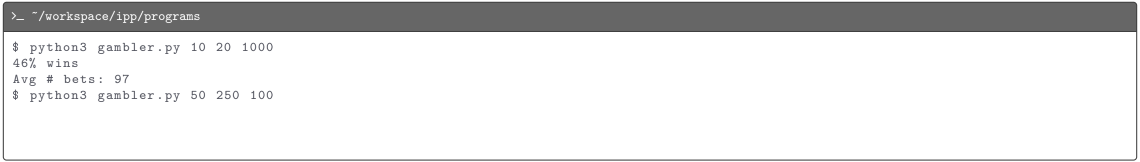
```
>_ ~/workspace/ipp/programs
```

```
$ python3 gambler.py 10 20 1000
46% wins
Avg # bets: 97
$ _
```

Applications

Program: `gambler.py`

- Command-line input: *stake* (int), *goal* (int), and *trials* (int)
- Standard output: percentage of wins and average number of bets per experiment



```
>_ ~/workspace/ipp/programs
```

```
$ python3 gambler.py 10 20 1000
46% wins
Avg # bets: 97
$ python3 gambler.py 50 250 100
```

Applications

Program: `gambler.py`

- Command-line input: *stake* (int), *goal* (int), and *trials* (int)
- Standard output: percentage of wins and average number of bets per experiment

```
>_ ~/workspace/ipp/programs

$ python3 gambler.py 10 20 1000
46% wins
Avg # bets: 97
$ python3 gambler.py 50 250 100
19% wins
Avg # bets: 12069
$ _
```

Applications

Applications

gambler.py

```python
import stdio
import sys
import stdrandom

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 0 < cash < goal:
        bets += 1
        if stdrandom.bernoulli():
            cash += 1
        else:
            cash -= 1
    if cash == goal:
        wins += 1
stdio.writeln(str(100 * wins // trials) + "% wins")
stdio.writeln("Avg # bets: " + str(bets // trials))
```

Applications

Applications

Program: `factors.py`

Applications

Program: `factors.py`

- Command-line input: n (int)

Applications

Program: `factors.py`

- Command-line input: n (int)
- Standard output: prime factors of n

Applications

Program: `factors.py`

- Command-line input: *n* (int)
- Standard output: prime factors of *n*

```
>_ ~/workspace/ipp/programs

$ _
```

Applications

Program: `factors.py`

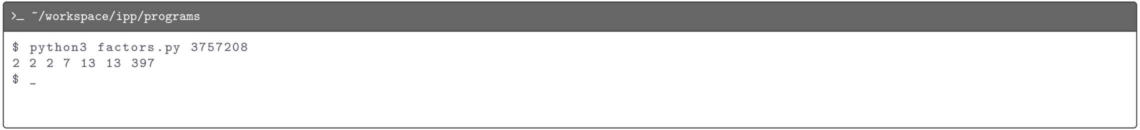
- Command-line input: *n* (int)
- Standard output: prime factors of *n*

```
>_ ~/workspace/ipp/programs

$ python3 factors.py 3757208
```

Applications

Program: `factors.py`

- Command-line input: *n* (int)
- Standard output: prime factors of *n*



```
$ python3 factors.py 3757208
2 2 2 7 13 13 397
$ _
```

Applications

Program: `factors.py`

- Command-line input: *n* (int)
- Standard output: prime factors of *n*

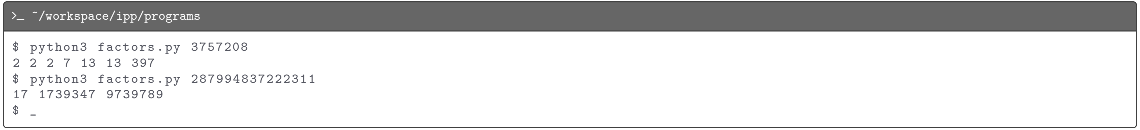
```
>_ ~/workspace/ipp/programs
```

```
$ python3 factors.py 3757208
2 2 2 7 13 13 397
$ python3 factors.py 287994837222311
```

Applications

Program: `factors.py`

- Command-line input: *n* (int)
- Standard output: prime factors of *n*



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

Applications

Applications

factors.py

```python
import stdio
import sys

n = int(sys.argv[1])
factor = 2
while factor * factor <= n:
    while n % factor == 0:
        stdio.write(str(factor) + " ")
        n //= factor
    factor += 1
if n > 1:
    stdio.write(n)
stdio.writeln()
```