

Understanding our first program



JAN 26 2015

Problem: Converting decimal numbers to binary



- Given a non-negative integer, convert it into its **binary equivalent**.
- **Example:**
 - **Input:** 123 **Output:** 1111011
 - **Input:** 1363 **Output:** 10101010011
 - **Input:** 12 **Output:** 1100

Our first program



```
n = int(input("Enter a positive integer:"))  
while n > 0:  
    print n % 2  
    n = n/2
```

Understanding the input statement



```
n = int(input("Type a nonnegative integer:"))
```

Assignment statement

- = is the **assignment operator**
- n is a **variable**
- The stuff on the right hand side is an **expression** that gets **evaluated** and its value gets **assigned** to the variable n

Examples of assignment statements



- $n = 9$

- $n = n/2$

(Assignment operator is not algebraic equality. “/” is “true division.”)

- $n = n + 1$

(A commonly used assignment statement for incrementing the variable n .)

- $m = n \% 2$

(m gets assigned 1 if n is odd; otherwise m gets assigned 0.)

- $m = n//5$

(“//” is “floor division.”).

The input function



`input(prompt)`

- This function is a *built-in* Python function and is always available.
- The **prompt** is written to output (screen) and then the function reads a line from input (keyboard) and *returns* what it reads.
- **prompt** is an *argument* to the function **input**.

Functions in Python



- When you are first taught (mathematical) functions in school, you are told to view them as *input-output machines*.
- This is a useful view for functions in Python also.
- The programmer *calls* a function with appropriate inputs, called *arguments* and the function does something (we may not know what) and produces an output.
- In Python, functions can be *built-in* (e.g., `input()`) or *user defined*.

input returns a string



Try this code snippet. What happens?

```
x = input("Enter a number:")  
x = x + 1
```

What the user types is read in as a string, the expression on the right hand side evaluates to a string and `x` gets assigned a string.

Data types in Python



- Every object (e.g., constants, variables) in Python has a *type*
- An object's type determines what operations can be performed on that object.
- Use the Python built-in function **type** to determine an object's data type.

Data types in Python



- Examples:

Constant

“Enter a number”

1034

55.0

type

string

integer

floating point

- Python has many *built-in* data types. For now we will work with four types:

type

integer

string

floating point

boolean

Python’s type name

int

str

float

bool

Type of a variable



- The type of a variable is the type of what it was most recently assigned.

Example:

```
x = 15
```

```
type(x)           int
```

```
x = x*1.0
```

```
type(x)           float
```

This ability of the same variable to have different types within a program is called *dynamic typing*.

Operators and data types



- The meaning of *operators* (e.g., +, //) depends on the data types they are operating on.

Expression	Value	Type
9//2	4	int
9.0//2	4.0	float
9/2.0	4.5	float
9/2	4.5	float
5 + 1	6	int
5 + 1.0	6.0	float
"hello, "+" friend"	"hello, friend"	string

Conversions between data types



- Python provides built-in functions for converting between data types.

- **Examples:**

Expression	Value
<code>int("320")</code>	320
<code>float("320")</code>	320.0
<code>str(134)</code>	"134"

Last slide on the first line



```
n = int(input("Enter a positive integer:"))
```

1. `input` prints the prompt, reads a line of the user's input, and returns what is read as a string.
2. The string returned by `input` gets converted to an integer by the function `int`.
3. This integer gets assigned to the variable `n`.

What is the value and type of each expression?



Expression	Value	Type
<code>10 + (12/2.0)</code>		
<code>"12" + "0"</code>		
<code>int("200")//10</code>		
<code>5 + 12/5</code>		
<code>str(25//4) + "00"</code>		
<code>9876 % 10</code>		
<code>str(9876 % 100)</code>		
<code>(12/5.0) + (12/5)</code>		

You'll get more practice in the discussion section and in Practice Problem Set 2.

On while-loops



```
Line 1
while boolean expression:
    Line 2
    Line 3
Line 4
```

- *while*-loops affect the *flow* of the program, i.e., the order in which program statements are executed. So *while*-loops are examples of a *control-flow* statements.
- For the above example the flow of the program is:

Line 1, bool expr (True), Line 2, Line 3, bool expr (True), Line 2, Line 3, bool expr (False), Line 4

Body of *while* loop



- Lines 2 and 3 form the *body* of the *while* loop
- Python uses indentation to identify the lines following the *while* statement that constitute the body of the *while* loop.

Boolean expressions



- Python has a type called `bool`
- The constants in this type are `True` and `False`.
(Not `true` and `false`!)

- The comparison operators:

`<` `>` `<=` `>=` `!=` `==`

can be used to construct *boolean expressions*, i.e., expressions that evaluate to `True` or `False`.

Boolean expressions: examples



- Suppose x has the value 10

Expression	Value	Type
$x < 10$	False	bool
$x \neq 100$	True	bool
$x \leq 10$	True	bool
$x > -10$	True	bool
$x \geq 11$	False	bool

Revisiting our program



```
n = int(input("Enter a positive integer:"))  
while n > 0:  
    print(n % 2)  
    n = n//2
```

- The boolean expression is **True** when n is positive and is **False** when n is less than or equal to 0.
- **Example:** Suppose n is initially 25. Then n takes on the values (in this order): 25, 12, 6, 3, 1, 0. When n becomes 0, the program exits the while-loop.