CS 115 Lecture 4

More Python; testing software

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

8 September 2015

Syntax: Statements

A **statement** is the smallest unit of code that can be executed on its own.

- So far we've seen:
  - Assignment: `sum = first + second`
  - Function call: `print("Hello")`

Syntax: Statements

A **statement** is the smallest unit of code that can be executed on its own.

- So far we've seen:
  - Assignment: `sum = first + second`
  - Function call: `print("Hello")`
- Usually one line, but **compound statements** can be bigger.
  - `def`, `for`, `if`, etc.
  - We'll see more of these in a few weeks.

Syntax: Statements

A **statement** is the smallest unit of code that can be executed on its own.

- So far we've seen:
  - Assignment: `sum = first + second`
  - Function call: `print("Hello")`
- Usually one line, but **compound statements** can be bigger.
  - `def`, `for`, `if`, etc.
  - We'll see more of these in a few weeks.
- Comments aren't statements: they can't be executed.

Syntax: Statements

A **statement** is the smallest unit of code that can be executed on its own.

- So far we've seen:
  - Assignment: `sum = first + second`
  - Function call: `print("Hello")`
- Usually one line, but **compound statements** can be bigger.
  - `def`, `for`, `if`, etc.
  - We'll see more of these in a few weeks.
- Comments aren't statements: they can't be executed.

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`
  - Variable name: `user_name`

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`
  - Variable name: `user_name`
  - Arithmetic: `3 * (x + 2)`

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`
  - Variable name: `user_name`
  - Arithmetic: `3 * (x + 2)`
    - ★ `(x + 2)` is itself an expression
    - ★ ...as are `x` and `2`

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`
  - Variable name: `user_name`
  - Arithmetic: `3 * (x + 2)`
    - `(x + 2)` is itself an expression
    - ...as are `x` and `2`
    - It's expressions all the way down!

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`
  - Variable name: `user_name`
  - Arithmetic: `3 * (x + 2)`
    - ★ `(x + 2)` is itself an expression
    - ★ ...as are `x` and `2`
    - ★ It's expressions all the way down!
  - Function call: `input("What is your name?")`

Syntax: Expressions

An **expression** is a piece of code that has a value. It is an even smaller and more fundamental unit than the statement.

- Something you could use on the right hand side of assignment.
- Can be used as part of bigger expressions.
- Examples:
  - Literals: `2`, `3.4`, `"Python"`
  - Variable name: `user_name`
  - Arithmetic: `3 * (x + 2)`
    - ★ `(x + 2)` is itself an expression
    - ★ ...as are `x` and `2`
    - ★ It's expressions all the way down!
  - Function call: `input("What is your name?")`

Semantics: Data types

Inside the computer, everything is just bits. A **data type** says how to interpret those bits, and what we can do with them.
Every expression in Python has a type. Some of the built-in types:

Type	Description	Examples
`int`	integer numbers	`2`, `-44`
`float`	floating-point numbers	`3.0`, `-0.9`, `6.022e23`
`bool`	Boolean (true-false) values	`True`, `False`
`str`	strings of characters	`'hello'`, `"0"`
`list`	lists of values	`[ "Valjean", 24601 ]`, `[ 2, 3, 5, 7 ]`

Semantics: Data types

Inside the computer, everything is just bits. A **data type** says how to interpret those bits, and what we can do with them.
Every expression in Python has a type. Some of the built-in types:

Type	Description	Examples
`int`	integer numbers	`2`, `-44`
`float`	floating-point numbers	`3.0`, `-0.9`, `6.022e23`
`bool`	Boolean (true-false) values	`True`, `False`
`str`	strings of characters	`'hello'`, `"0"`
`list`	lists of values	`[ "Valjean", 24601 ]`, `[ 2, 3, 5, 7 ]`

Integers

Type `int` represents **integers**: whole numbers that may be positive, negative, or zero.

- Literal integers: a sequence of digits: 24601

Integers

Type `int` represents **integers**: whole numbers that may be positive, negative, or zero.

- Literal integers: a sequence of digits: 24601
  - ...with no leading zeros!
  - 0 is okay, 007 is not.

Integers

Type `int` represents **integers**: whole numbers that may be positive, negative, or zero.

- Literal integers: a sequence of digits: 24601
  - ...with no leading zeros!
  - 0 is okay, 007 is not.
- In Python, integers have no limit to their size.
  - As many digits as you have memory for.
  - That isn't true in most languages, like C++ and Java.

Integers

Type `int` represents **integers**: whole numbers that may be positive, negative, or zero.

- Literal integers: a sequence of digits: 24601
  - ...with no leading zeros!
  - 0 is okay, 007 is not.
- In Python, integers have no limit to their size.
  - As many digits as you have memory for.
  - That isn't true in most languages, like C++ and Java.

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

- Limited precision and range.

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

- Limited precision and range.
- Two forms of literal floats:
  - Number with a decimal point:
    `3.14`, `.027`, `0.001`, `3.`, `1.0`

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

- Limited precision and range.
- Two forms of literal floats:
  - Number with a decimal point:
    `3.14`, `.027`, `0.001`, `3.`, `1.0`
    - Must have a decimal point!
    - `1.0` or `1.` is a float, but `1` is an integer!

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

- Limited precision and range.
- Two forms of literal floats:
  - Number with a decimal point:
    `3.14`, `.027`, `0.001`, `3.`, `1.0`
    - Must have a decimal point!
    - `1.0` or `1.` is a float, but `1` is an integer!
  - "E"-notation (scientific notation):
    - `6.022e23`, `1.0E9`, `31e-2`

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

- Limited precision and range.
- Two forms of literal floats:
  - Number with a decimal point:
    `3.14`, `.027`, `0.001`, `3.`, `1.0`
    - Must have a decimal point!
    - `1.0` or `1.` is a float, but `1` is an integer!
  - "E"-notation (scientific notation):
    - `6.022e23`, `1.0E9`, `31e-2`
    - Write `e` for "times 10 to the"
    - Does not need a decimal point: the `e` is enough.
    - Exponent must be an integer.

Floating-point

Type `float` represents **floating-point numbers**: numbers with a decimal point.

- Limited precision and range.
- Two forms of literal floats:
  - Number with a decimal point:
    `3.14`, `.027`, `0.001`, `3.`, `1.0`
    - Must have a decimal point!
    - `1.0` or `1.` is a float, but `1` is an integer!
  - "E"-notation (scientific notation):
    - `6.022e23`, `1.0E9`, `31e-2`
    - Write `e` for "times 10 to the"
    - Does not need a decimal point: the `e` is enough.
    - Exponent must be an integer.
- In some languages these are called "doubles"

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - ★ (leading and trailing zeros don't count)

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - ★ (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`—gives `inf` (infinity)

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`—gives `inf` (infinity)
  - Try: `1e-324`

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`—gives `inf` (infinity)
  - Try: `1e-324`—gives `0.0`

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`—gives `inf` (infinity)
  - Try: `1e-324`—gives `0.0`
- The exact limits are on the number of bits, not digits.
  - Even `0.1` can't be represented exactly
    - Try: `0.1 + 0.1 + 0.1`

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`—gives `inf` (infinity)
  - Try: `1e-324`—gives `0.0`
- The exact limits are on the number of bits, not digits.
  - Even `0.1` can't be represented exactly
    - Try: `0.1 + 0.1 + 0.1`—gives `0.30000000000000004`

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary).
  - **Exponent**: how far to move the decimal (binary) point.
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count)
  - Limits the **precision** of a float.
  - Try: `10000000000000002.0 - 10000000000000001.0`
    - Python's answer is `2.0`: the 1 was rounded down!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: `1e309`—gives `inf` (infinity)
  - Try: `1e-324`—gives `0.0`
- The exact limits are on the number of bits, not digits.
  - Even `0.1` can't be represented exactly
    - Try: `0.1 + 0.1 + 0.1`—gives `0.30000000000000004`

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (+ - * **) the rules are:

- If both operands are ints, the result is an int.

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (`+ - * **`) the rules are:

- If both operands are ints, the result is an int.
  - `3 + 5` $\rightarrow$ `8`
  - `2 ** 100` $\rightarrow$ `1267650600228229401496703205376`

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (`+ - * **`) the rules are:

- If both operands are ints, the result is an int.
  - `3 + 5` → `8`
  - `2 ** 100` → `1267650600228229401496703205376`
- If one or both operand is a float, the result is a float.

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (`+ - * **`) the rules are:

- If both operands are ints, the result is an int.
  - `3 + 5` → `8`
  - `2 ** 100` → `1267650600228229401496703205376`
- If one or both operand is a float, the result is a float.
  - `3.0 + 0.14` → `3.14`
  - `100 - 1.0` → `99.0`
  - `2.0 ** 100` → `1.2676506002282294e+30`

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (`+ - * **`) the rules are:

- If both operands are ints, the result is an int.
  - `3 + 5` → `8`
  - `2 ** 100` → `1267650600228229401496703205376`
- If one or both operand is a float, the result is a float.
  - `3.0 + 0.14` → `3.14`
  - `100 - 1.0` → `99.0`
  - `2.0 ** 100` → `1.2676506002282294e+30`
- There is one exception...

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (`+ - * **`) the rules are:

- If both operands are ints, the result is an int.
  - `3 + 5` → `8`
  - `2 ** 100` → `1267650600228229401496703205376`
- If one or both operand is a float, the result is a float.
  - `3.0 + 0.14` → `3.14`
  - `100 - 1.0` → `99.0`
  - `2.0 ** 100` → `1.2676506002282294e+30`
- There is one exception...
  - What is `1/2` ?

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (`+ - * **`) the rules are:

- If both operands are ints, the result is an int.
  - `3 + 5` → `8`
  - `2 ** 100` → `1267650600228229401496703205376`
- If one or both operand is a float, the result is a float.
  - `3.0 + 0.14` → `3.14`
  - `100 - 1.0` → `99.0`
  - `2.0 ** 100` → `1.2676506002282294e+30`
- There is one exception...
  - What is `1/2` ?

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.
  - If both operands are integers, so is the result.

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.
  - If both operands are integers, so is the result.
    - `22 // 7` → 3
    - `1 // 2` → 0

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.
  - If both operands are integers, so is the result.
    - `22 // 7` → 3
    - `1 // 2` → 0
  - If either operand is a float, so is the result.
    - But it still has a whole-number value.

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.
  - If both operands are integers, so is the result.
    - `22 // 7` → `3`
    - `1 // 2` → `0`
  - If either operand is a float, so is the result.
    - But it still has a whole-number value.
    - `22 // 7.0` → `3.0`
    - `3.1 // 0.5` → `6.0`

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.
  - If both operands are integers, so is the result.
    - `22 // 7` → 3
    - `1 // 2` → 0
  - If either operand is a float, so is the result.
    - But it still has a whole-number value.
    - `22 // 7.0` → `3.0`
    - `3.1 // 0.5` → `6.0`
- In either case, dividing by zero is a run-time error!

Division

Python actually has *two* division operators, / and //.

- / *always* gives a float.
  - `1 / 2` → `0.5`
  - `6 / 3` → `2.0`
  - `3.0 / 0.5` → `6.0`
- // does **floor division**: rounds the answer down to a whole number.
  - If both operands are integers, so is the result.
    - `22 // 7` → 3
    - `1 // 2` → 0
  - If either operand is a float, so is the result.
    - But it still has a whole-number value.
    - `22 // 7.0` → `3.0`
    - `3.1 // 0.5` → `6.0`
- In either case, dividing by zero is a run-time error!

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).
  - 6 % 3 $\rightarrow$ 0
  - 7 % 3 $\rightarrow$ 1
  - 8 % 3 $\rightarrow$ 2
  - 9 % 3 $\rightarrow$ 0

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).
  - `6 % 3` → `0`
  - `7 % 3` → `1`
  - `8 % 3` → `2`
  - `9 % 3` → `0`
- Uses of modulo:
  - Even/odd: `n` is even if `n % 2` is zero.

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).
  - `6 % 3` → `0`
  - `7 % 3` → `1`
  - `8 % 3` → `2`
  - `9 % 3` → `0`
- Uses of modulo:
  - Even/odd: `n` is even if `n % 2` is zero.
  - Digits: `n % 10` is the last digit of `n`.

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).
  - `6 % 3` → 0
  - `7 % 3` → 1
  - `8 % 3` → 2
  - `9 % 3` → 0
- Uses of modulo:
  - Even/odd: `n` is even if `n % 2` is zero.
  - Digits: `n % 10` is the last digit of `n`.
  - "Clock arithmetic"
    - Minutes are mod 60: 3:**58** + **15** minutes = 4:**13**
    - Hours are mod 12: **10**:00 + **4** hours = **2**:00

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).
  - `6 % 3` → `0`
  - `7 % 3` → `1`
  - `8 % 3` → `2`
  - `9 % 3` → `0`
- Uses of modulo:
  - Even/odd: `n` is even if `n % 2` is zero.
  - Digits: `n % 10` is the last digit of `n`.
  - "Clock arithmetic"
    - Minutes are mod 60: 3:**58** + **15** minutes = 4:**13**
    - Hours are mod 12: **10**:00 + **4** hours = **2**:00
- Python can do modulo on floats.
  - `5 % 2.4` → `0.2`
  - Far more common with ints, though.

Remainder (modulo)

The % operator (**modulo** or **mod**) finds the remainder of a division.

- Between 0 (inclusive) and the right hand side (exclusive).
  - `6 % 3` → `0`
  - `7 % 3` → `1`
  - `8 % 3` → `2`
  - `9 % 3` → `0`
- Uses of modulo:
  - Even/odd: `n` is even if `n % 2` is zero.
  - Digits: `n % 10` is the last digit of `n`.
  - "Clock arithmetic"
    - Minutes are mod 60: 3:**58** + **15** minutes = 4:**13**
    - Hours are mod 12: **10**:00 + **4** hours = **2**:00
- Python can do modulo on floats.
  - `5 % 2.4` → `0.2`
  - Far more common with ints, though.

Booleans

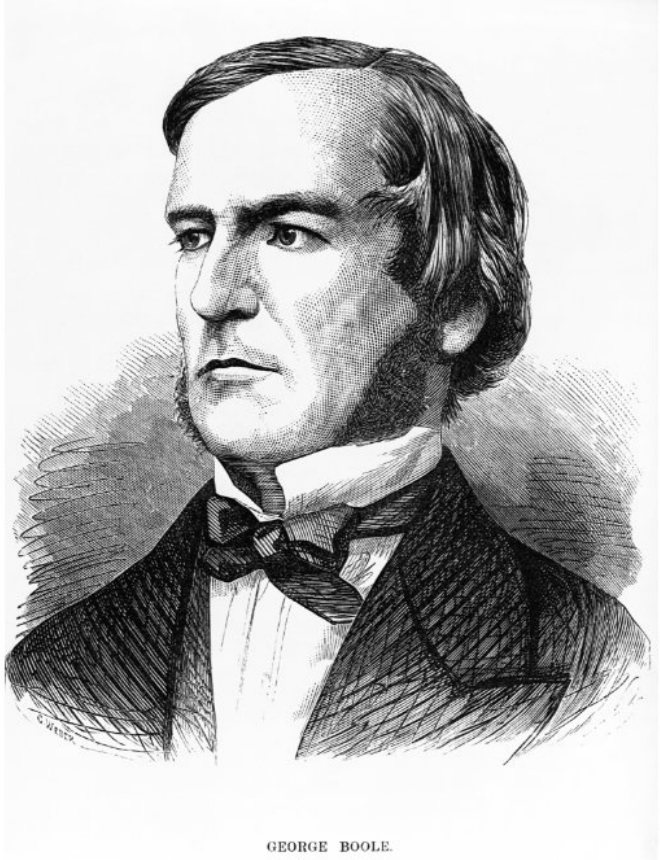
Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.

Booleans

Type b

- N



George Boole, English computer scientist.

Image: Computer History Museum

Booleans

Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.
- Exactly two values: `True`, `False`

Booleans

Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.
- Exactly two values: `True`, `False`
  - No quotes! They aren't strings.
  - Case-sensitive as always: capital T and F.

Booleans

Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.
- Exactly two values: `True`, `False`
  - No quotes! They aren't strings.
  - Case-sensitive as always: capital T and F.
- Boolean values are the basis of computer circuits: EE 280.

Booleans

Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.
- Exactly two values: `True`, `False`
  - No quotes! They aren't strings.
  - Case-sensitive as always: capital T and F.
- Boolean values are the basis of computer circuits: EE 280.
- Can't do arithmetic on them.
  - Can do `and`, `or`, and `not`.

Booleans

Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.
- Exactly two values: `True`, `False`
  - No quotes! They aren't strings.
  - Case-sensitive as always: capital T and F.
- Boolean values are the basis of computer circuits: EE 280.
- Can't do arithmetic on them.
  - Can do `and`, `or`, and `not`.
  - Most often used with `if` and `while` statements.
  - More about boolean operations in chapter 4.

Booleans

Type `bool` represents **boolean values**.

- Named after George Boole, English mathematician and logician.
- Exactly two values: `True`, `False`
  - No quotes! They aren't strings.
  - Case-sensitive as always: capital T and F.
- Boolean values are the basis of computer circuits: EE 280.
- Can't do arithmetic on them.
  - Can do `and`, `or`, and `not`.
  - Most often used with `if` and `while` statements.
  - More about boolean operations in chapter 4.

Strings

Type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.

Strings

Type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.
  - `'hello'`, `"world"`, `""`
  - Use whichever quote isn't in the string:

    `'some "quotes"'`, `"a'postrophe"`

Strings

Type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.
  - `'hello'`, `"world"`, `""`
  - Use whichever quote isn't in the string:

    ```
    'some "quotes"', "a'postrophe"
    ```
  - If you have to include the quote character, **escape** it with a backslash:

    ```
    msg = 'the word "don\'t" is 5 characters long'
    ```
  - Have to escape backslashes, too:

    ```
    folder = "C:\\Python 3.4"
    ```

Strings

Type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.
  - `'hello'`, `"world"`, `""`
  - Use whichever quote isn't in the string:
    `'some "quotes"'`, `"a'postrophe"`
  - If you have to include the quote character, **escape** it with a backslash:
    `msg = 'the word "don\'t" is 5 characters long'`
  - Have to escape backslashes, too:
    `folder = "C:\\Python 3.4"`
  - Special characters: tab `\t`, newline `\n`.

Strings

Type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.
  - `'hello'`, `"world"`, `""`
  - Use whichever quote isn't in the string:
    `'some "quotes"'`, `"a'postrophe"`
  - If you have to include the quote character, **escape** it with a backslash:
    `msg = 'the word "don\'t" is 5 characters long'`
  - Have to escape backslashes, too:
    `folder = "C:\\Python 3.4"`
  - Special characters: tab `\t`, newline `\n`.
- Can perform a few operations on strings:
  - **Concatenate** (join) strings with +:
    `greeting = "Hello, " + name`

Strings

Type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.
  - `'hello'`, `"world"`, `""`
  - Use whichever quote isn't in the string:
    ```
    'some "quotes"', "a'postrophe"
    ```
  - If you have to include the quote character, **escape** it with a backslash:
    ```
    msg = 'the word "don\'t" is 5 characters long'
    ```
  - Have to escape backslashes, too:
    ```
    folder = "C:\\Python 3.4"
    ```
  - Special characters: tab `\t`, newline `\n`.
- Can perform a few operations on strings:
  - **Concatenate** (join) strings with +:
    ```
    greeting = "Hello, " + name
    ```
  - Repeat a string by "multiplying" with an integer:
    ```
    rating = '⋆' * 4    # ⋆⋆⋆⋆
    bird = 2 * 'do'     # dodo
    ```

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses:
  - `float(2)` $\rightarrow$ `2.0`
  - `int(3.14)` $\rightarrow$ `3`
  - `str(1.2e3)` $\rightarrow$ `"1200.0"`
  - `int("02")` $\rightarrow$ `2`
  - `float("0")` $\rightarrow$ `0.0`

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses:
  - `float(2)` $\rightarrow$ `2.0`
  - `int(3.14)` $\rightarrow$ `3`
  - `str(1.2e3)` $\rightarrow$ `"1200.0"`
  - `int("02")` $\rightarrow$ `2`
  - `float("0")` $\rightarrow$ `0.0`
- Converting double to int rounds towards zero ($+$ down, $-$ up)

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses:
  - `float(2)` $\rightarrow$ `2.0`
  - `int(3.14)` $\rightarrow$ `3`
  - `str(1.2e3)` $\rightarrow$ `"1200.0"`
  - `int("02")` $\rightarrow$ `2`
  - `float("0")` $\rightarrow$ `0.0`
- Converting double to int rounds towards zero ($+$ down, $-$ up)
- Run-time error if a string could not be converted:
  - `n = int("hello")  # CRASHES with ValueError`

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses:
  - `float(2)` → `2.0`
  - `int(3.14)` → `3`
  - `str(1.2e3)` → `"1200.0"`
  - `int("02")` → `2`
  - `float("0")` → `0.0`
- Converting double to int rounds towards zero (+ down, − up)
- Run-time error if a string could not be converted:
  - `n = int("hello")  # CRASHES with ValueError`
  - `p = int("3.2")  # CRASHES, but int(float("3.2")) is OK`

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses:
  - `float(2)` → `2.0`
  - `int(3.14)` → `3`
  - `str(1.2e3)` → `"1200.0"`
  - `int("02")` → `2`
  - `float("0")` → `0.0`
- Converting double to int rounds towards zero (+ down, − up)
- Run-time error if a string could not be converted:
  - `n = int("hello")  # CRASHES with ValueError`
  - `p = int("3.2")  # CRASHES, but int(float("3.2")) is OK`
- Converting a string **does not** do arithmetic:
  - `half = float("1/2")  # CRASHES`

Converting between types

Converting between types is also called **type casting**.

- Write the name of the type you are converting to, then the expression to convert in parentheses:
  - `float(2)` → `2.0`
  - `int(3.14)` → `3`
  - `str(1.2e3)` → `"1200.0"`
  - `int("02")` → `2`
  - `float("0")` → `0.0`
- Converting double to int rounds towards zero (+ down, − up)
- Run-time error if a string could not be converted:
  - `n = int("hello")  # CRASHES with ValueError`
  - `p = int("3.2")  # CRASHES, but int(float("3.2")) is OK`
- Converting a string **does not** do arithmetic:
  - `half = float("1/2")  # CRASHES`

Output: `print`

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.

Output: `print`

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.

Output: `print`

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.
  - Each argument can be a literal, variable, expression, ...
  - Arguments can be any type: string, integer, float, ...

Output: `print`

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.
  - Each argument can be a literal, variable, expression, ...
  - Arguments can be any type: string, integer, float, ...
    - `print("Welcome to my program")`

Output: `print`

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.
  - Each argument can be a literal, variable, expression, ...
  - Arguments can be any type: string, integer, float, ...
    - `print("Welcome to my program")`
    - `print(6 * 7)`

Output: print

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.
  - Each argument can be a literal, variable, expression, ...
  - Arguments can be any type: string, integer, float, ...
    - `print("Welcome to my program")`
    - `print(6 * 7)`
    - `print("Hello", name)`

Output: print

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.
  - Each argument can be a literal, variable, expression, ...
  - Arguments can be any type: string, integer, float, ...
    - `print("Welcome to my program")`
    - `print(6 * 7)`
    - `print("Hello", name)`
    - `print()`

Output: print

Every program needs to do output of some kind: to the screen, a file, etc.
In Python, we use the **print** function.

- Sends output to "standard output".
  - This is usually the shell window.
  - Or the window that appears when you double-click a Python program.
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print.
    - Can have zero, one, or more arguments.
  - Each argument can be a literal, variable, expression, ...
  - Arguments can be any type: string, integer, float, ...
    - `print("Welcome to my program")`
    - `print(6 * 7)`
    - `print("Hello", name)`
    - `print()`

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.
- If multiple arguments are given, puts a space between.

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.
- If multiple arguments are given, puts a space between.
- Outputs a "newline" character at the end.
  - Moves the cursor to the beginning of the next line.

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.
- If multiple arguments are given, puts a space between.
- Outputs a "newline" character at the end.
  - Moves the cursor to the beginning of the next line.
  - No-argument `print()` prints just a newline.

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.
- If multiple arguments are given, puts a space between.
- Outputs a "newline" character at the end.
  - Moves the cursor to the beginning of the next line.
  - No-argument `print()` prints just a newline.
- The `print` function does not return a value.
  - That means you can't (usefully) use it in an expression:

```
x = print(2)  # BAD
```

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.
- If multiple arguments are given, puts a space between.
- Outputs a "newline" character at the end.
  - Moves the cursor to the beginning of the next line.
  - No-argument `print()` prints just a newline.
- The `print` function does not return a value.
  - That means you can't (usefully) use it in an expression:

    ```
    x = print(2)  # BAD
    ```

    - ★ This isn't a syntax error, but `x`'s value will be `None`.
    - ★ Not very useful: usually a semantic error.

Semantics of `print`

- Evaluates the arguments (computes their values).
- Prints values to standard output, starting at the cursor.
- If multiple arguments are given, puts a space between.
- Outputs a "newline" character at the end.
  - Moves the cursor to the beginning of the next line.
  - No-argument `print()` prints just a newline.
- The `print` function does not return a value.
  - That means you can't (usefully) use it in an expression:

    ```
    x = print(2)  # BAD
    ```

    - ★ This isn't a syntax error, but `x`'s value will be `None`.
    - ★ Not very useful: usually a semantic error.

Extra arguments to `print`

Sometimes you don't want spaces between the arguments, or don't want a newline at the end.

Extra arguments to `print`

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- `sep=`*`string`*: Use *`string`* to separate arguments instead of space.

Extra arguments to `print`

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- `sep=`*`string`*: Use *`string`* to separate arguments instead of space.
  - `print(month, day, year, sep = "/")`
    - Might output: `1/27/2015`

Extra arguments to `print`

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- `sep=`*`string`*: Use *`string`* to separate arguments instead of space.
  - `print(month, day, year, sep = "/")`
    - Might output: `1/27/2015`
- `end=`*`string`*: Print *`string`* at the end, not newline.
  - That means the next print will start on the same line:
    - ```
      print("The answer is", end=":")
      print(answer)
      ```
      ```

Extra arguments to print

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- sep=*string*: Use *string* to separate arguments instead of space.
  - `print(month, day, year, sep = "/")`
    - Might output: `1/27/2015`
- end=*string*: Print *string* at the end, not newline.
  - That means the next print will start on the same line:
    - ```
      print("The answer is", end=":")
      print(answer)
      ```
    - Output: `The answer is:42`
- Either string can be empty:
  - `print(f_initial, m_initial, l_initial, sep='')`
      ```

Extra arguments to print

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- sep=*string*: Use *string* to separate arguments instead of space.
  - `print(month, day, year, sep = "/")`
    - Might output: `1/27/2015`
- end=*string*: Print *string* at the end, not newline.
  - That means the next print will start on the same line:
    - ```
      print("The answer is", end=":")
      print(answer)
      ```
    - Output: `The answer is:42`
- Either string can be empty:
  - `print(f_initial, m_initial, l_initial, sep='')`
  - Output: `NFM`
      ```

Extra arguments to print

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- sep=*string*: Use *string* to separate arguments instead of space.
  - `print(month, day, year, sep = "/")`
    - Might output: `1/27/2015`
- end=*string*: Print *string* at the end, not newline.
  - That means the next print will start on the same line:
    - ```
      print("The answer is", end=":")
      print(answer)
      ```
    - Output: `The answer is:42`
- Either string can be empty:
  - `print(f_initial, m_initial, l_initial, sep='')`
  - Output: `NFM`
- Can use both, but keyword arguments must come at the end!
      ```

Extra arguments to print

Sometimes you don't want spaces between the arguments,
or don't want a newline at the end.

- You can control these with so-called **keyword arguments**.
- sep=*string*: Use *string* to separate arguments instead of space.
  - `print(month, day, year, sep = "/")`
    - Might output: `1/27/2015`
- end=*string*: Print *string* at the end, not newline.
  - That means the next print will start on the same line:
    - ```
      print("The answer is", end=":")
      print(answer)
      ```
    - Output: `The answer is:42`
- Either string can be empty:
  - `print(f_initial, m_initial, l_initial, sep='')`
  - Output: `NFM`
- Can use both, but keyword arguments must come at the end!
      ```

Input

Most programs also need to get input from the user.
In Python we do this with `input` function.

- Syntax: `input(prompt)`
  - One argument (unlike print)
  - The argument is optional: `input()`

Input

Most programs also need to get input from the user.
In Python we do this with `input` function.

- Syntax: `input(prompt)`
  - One argument (unlike print)
  - The argument is optional: `input()`
- **Returns** (evaluates to) a string.
  - Usually used with assignment:

```
name = input("What is your name?  ")
```

Input

Most programs also need to get input from the user.
In Python we do this with `input` function.

- Syntax: `input(prompt)`
  - One argument (unlike print)
  - The argument is optional: `input()`
- **Returns** (evaluates to) a string.
  - Usually used with assignment:

```
name = input("What is your name?  ")
```

Semantics of `input`

- The `input` function first prints the prompt.
  - Without adding a newline! Usually ends in a space:

```
name = input("What is your name?  ")
```

Semantics of `input`

- The `input` function first prints the prompt.
  - Without adding a newline! Usually ends in a space:
    ```
    name = input("What is your name?  ")
    ```
  - Include a newline `\n` to get input on its own line:
    ```
    name = input("What is your name?\n")
    ```

Semantics of `input`

- The `input` function first prints the prompt.
  - Without adding a newline! Usually ends in a space:
    ```
    name = input("What is your name?  ")
    ```
  - Include a newline `\n` to get input on its own line:
    ```
    name = input("What is your name?\n")
    ```
  - If no prompt was given, prints nothing.

Semantics of `input`

- The `input` function first prints the prompt.
  - Without adding a newline! Usually ends in a space:
    ```
    name = input("What is your name?  ")
    ```
  - Include a newline `\n` to get input on its own line:
    ```
    name = input("What is your name?\n")
    ```
  - If no prompt was given, prints nothing.
- Pauses the execution of the program, displays a blinking cursor.
  - Waits for the user to press **Enter**.

Semantics of `input`

- The `input` function first prints the prompt.
  - Without adding a newline! Usually ends in a space:
    ```
    name = input("What is your name?  ")
    ```
  - Include a newline `\n` to get input on its own line:
    ```
    name = input("What is your name?\n")
    ```
  - If no prompt was given, prints nothing.
- Pauses the execution of the program, displays a blinking cursor.
  - Waits for the user to press **Enter**.
- Returns the entire line of input.
  - If the user pressed just **Enter**, returns an empty string.

Semantics of `input`

- The `input` function first prints the prompt.
  - Without adding a newline! Usually ends in a space:
    ```
    name = input("What is your name?  ")
    ```
  - Include a newline `\n` to get input on its own line:
    ```
    name = input("What is your name?\n")
    ```
  - If no prompt was given, prints nothing.
- Pauses the execution of the program, displays a blinking cursor.
  - Waits for the user to press **Enter**.
- Returns the entire line of input.
  - If the user pressed just **Enter**, returns an empty string.

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

```
name = input("What is your name?  ")
```

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

    ```
    name = input("What is your name?  ")
    ```

  - If you don't, throws away the input!

    ```
    input("Press Enter to continue.")
    ```

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

    ```
    name = input("What is your name?  ")
    ```

  - If you don't, throws away the input!

    ```
    input("Press Enter to continue.")
    ```

- What if you need a number instead?

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

    ```
    name = input("What is your name?  ")
    ```

  - If you don't, throws away the input!

    ```
    input("Press Enter to continue.")
    ```

- What if you need a number instead?
  - Combine it with type casting!

    ```
    age = int(input("How old are you?  "))
    ```

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

    ```
    name = input("What is your name?  ")
    ```

  - If you don't, throws away the input!

    ```
    input("Press Enter to continue.")
    ```

- What if you need a number instead?
  - Combine it with type casting!

    ```
    age = int(input("How old are you?  "))
    temp = float(input("What is the temperature?  "))
    ```

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

    ```
    name = input("What is your name?  ")
    ```
  - If you don't, throws away the input!

    ```
    input("Press Enter to continue.")
    ```
- What if you need a number instead?
  - Combine it with type casting!

    ```
    age = int(input("How old are you?  "))
    temp = float(input("What is the temperature?  "))
    ```
  - What if the input cannot be converted?

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

```
name = input("What is your name?  ")
```

  - If you don't, throws away the input!

```
input("Press Enter to continue.")
```

- What if you need a number instead?
  - Combine it with type casting!

```
age = int(input("How old are you?  "))
temp = float(input("What is the temperature?  "))
```

  - What if the input cannot be converted?
    - ★ Run-time error (`ValueError` exception)

Using input

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

```
name = input("What is your name?  ")
```

  - If you don't, throws away the input!

```
input("Press Enter to continue.")
```

- What if you need a number instead?
  - Combine it with type casting!

```
age = int(input("How old are you?  "))
temp = float(input("What is the temperature?  "))
```

  - What if the input cannot be converted?
    - ★ Run-time error (`ValueError` exception)

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!
- Verify that the progam:
  - Gives the correct outputs.
  - Doesn't crash unexpectedly.
  - Doesn't run forever (infinite loop).

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!
- Verify that the progam:
  - Gives the correct outputs.
  - Doesn't crash unexpectedly.
  - Doesn't run forever (infinite loop).
- For a four- or five-line program, could verify by inspection.

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!
- Verify that the progam:
  - Gives the correct outputs.
  - Doesn't crash unexpectedly.
  - Doesn't run forever (infinite loop).
- For a four- or five-line program, could verify by inspection.
  - But once it gets longer than that, needs *systematic* testing.

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!
- Verify that the progam:
  - Gives the correct outputs.
  - Doesn't crash unexpectedly.
  - Doesn't run forever (infinite loop).
- For a four- or five-line program, could verify by inspection.
  - But once it gets longer than that, needs *systematic* testing.
- Could plug in some random value and check the output.
  - But what if we missed something?

Testing

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!
- Verify that the progam:
  - Gives the correct outputs.
  - Doesn't crash unexpectedly.
  - Doesn't run forever (infinite loop).
- For a four- or five-line program, could verify by inspection.
  - But once it gets longer than that, needs *systematic* testing.
- Could plug in some random value and check the output.
  - But what if we missed something?
  - Need a plan...

Test cases

We will test our programs by trying out a number of **test cases**.

Test cases

We will test our programs by trying out a number of **test cases**.

- A typical test case has four parts:
  - Description: what are you testing for?
  - Input data you will give to the program.
  - The expected output or outcome from that input.
  - The actual output or outcome from that input.

Test cases

We will test our programs by trying out a number of **test cases**.

- A typical test case has four parts:
  - Description: what are you testing for?
  - Input data you will give to the program.
  - The expected output or outcome from that input.
  - The actual output or outcome from that input.
- Do the first three parts **before** writing the program.
  - Then fill out the actual output by running the program.

Test cases

We will test our programs by trying out a number of **test cases**.

- A typical test case has four parts:
  - Description: what are you testing for?
  - Input data you will give to the program.
  - The expected output or outcome from that input.
  - The actual output or outcome from that input.
- Do the first three parts **before** writing the program.
  - Then fill out the actual output by running the program.
  - In a software company, the last step is often done by dedicated testers (not the author).

Test cases

We will test our programs by trying out a number of **test cases**.

- A typical test case has four parts:
  - Description: what are you testing for?
  - Input data you will give to the program.
  - The expected output or outcome from that input.
  - The actual output or outcome from that input.
- Do the first three parts **before** writing the program.
  - Then fill out the actual output by running the program.
  - In a software company, the last step is often done by dedicated testers (not the author).
  - In CS 115, we'll usually omit the "actual output".
    - If it's different from the expected output, you have a bug.
    - Fix the bugs before turning in the program.

Test cases

We will test our programs by trying out a number of **test cases**.

- A typical test case has four parts:
  - Description: what are you testing for?
  - Input data you will give to the program.
  - The expected output or outcome from that input.
  - The actual output or outcome from that input.
- Do the first three parts **before** writing the program.
  - Then fill out the actual output by running the program.
  - In a software company, the last step is often done by dedicated testers (not the author).
  - In CS 115, we'll usually omit the "actual output".
    - If it's different from the expected output, you have a bug.
    - Fix the bugs before turning in the program.
    - At the very least, document the bug with a comment.

Test cases

We will test our programs by trying out a number of **test cases**.

- A typical test case has four parts:
  - Description: what are you testing for?
  - Input data you will give to the program.
  - The expected output or outcome from that input.
  - The actual output or outcome from that input.
- Do the first three parts **before** writing the program.
  - Then fill out the actual output by running the program.
  - In a software company, the last step is often done by dedicated testers (not the author).
  - In CS 115, we'll usually omit the "actual output".
    - If it's different from the expected output, you have a bug.
    - Fix the bugs before turning in the program.
    - At the very least, document the bug with a comment.

Test plan

A **test plan** is a table with a number of test cases.

- Quality is more important than quantity!
- Test cases shouldn't overlap too much.
  - If all your tests use positive numbers, how will you know whether negative numbers work?

Test plan

A **test plan** is a table with a number of test cases.

- Quality is more important than quantity!
- Test cases shouldn't overlap too much.
  - If all your tests use positive numbers, how will you know whether negative numbers work?
- Making a good test plan requires thought and attention to the problem specifications.

Test plan

A **test plan** is a table with a number of test cases.

- Quality is more important than quantity!
- Test cases shouldn't overlap too much.
  - If all your tests use positive numbers, how will you know whether negative numbers work?
- Making a good test plan requires thought and attention to the problem specifications.
- You should identify and test:
  - Normal cases.
  - Special cases.
  - Boundary cases.
  - Error cases.

Test plan

A **test plan** is a table with a number of test cases.

- Quality is more important than quantity!
- Test cases shouldn't overlap too much.
  - If all your tests use positive numbers, how will you know whether negative numbers work?
- Making a good test plan requires thought and attention to the problem specifications.
- You should identify and test:
  - Normal cases.
  - Special cases.
  - Boundary cases.
  - Error cases.

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.
Refund	Q	Q	R	-	

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.
Refund	Q	Q	R	-	Return two quarters.

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.
Refund	Q	Q	R	-	Return two quarters.
Refund, no money	R	-	-	-	

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.
Refund	Q	Q	R	-	Return two quarters.
Refund, no money	R	-	-	-	Do nothing.

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.
Refund	Q	Q	R	-	Return two quarters.
Refund, no money	R	-	-	-	Do nothing.
Even more test cases could be described...					

Sample test plan

Suppose you are writing code for a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Coke, return one quarter.
Not enough	Q	Q	C	-	Flash "0.75".
Continue	Q	C	D	C	Flash "0.75", vend one Coke, return two quarters.
Refund	Q	Q	R	-	Return two quarters.
Refund, no money	R	-	-	-	Do nothing.
Even more test cases could be described...					

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!
- This is a common source of errors in programming.
  - "Fencepost errors" or "off-by-one errors".

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!
- This is a common source of errors in programming.
  - "Fencepost errors" or "off-by-one errors".
- Whenever your program involves ranges (1–10, letters "L"–"R"):
  - Test the **boundary cases**.

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!
- This is a common source of errors in programming.
  - "Fencepost errors" or "off-by-one errors".
- Whenever your program involves ranges (1–10, letters "L"–"R"):
  - Test the **boundary cases**.
  - Not just the endpoints, also adjacent numbers.
    - So 0, 1, 2 (lower boundary), 9, 10, 11 (upper).
    - "K", "L", "M" and "Q", "R", "S".

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!
- This is a common source of errors in programming.
  - "Fencepost errors" or "off-by-one errors".
- Whenever your program involves ranges (1–10, letters "L"–"R"):
  - Test the **boundary cases**.
  - Not just the endpoints, also adjacent numbers.
    - So 0, 1, 2 (lower boundary), 9, 10, 11 (upper).
    - "K", "L", "M" and "Q", "R", "S".
- Why test boundary cases?
  - It's easy to miss one endpoint.
  - Or to go too far, past the endpoint.

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!
- This is a common source of errors in programming.
  - "Fencepost errors" or "off-by-one errors".
- Whenever your program involves ranges (1–10, letters "L"–"R"):
  - Test the **boundary cases**.
  - Not just the endpoints, also adjacent numbers.
    - So 0, 1, 2 (lower boundary), 9, 10, 11 (upper).
    - "K", "L", "M" and "Q", "R", "S".
- Why test boundary cases?
  - It's easy to miss one endpoint.
  - Or to go too far, past the endpoint.
  - Make sure in-range inputs are accepted.
  - Make sure out-of-range inputs are rejected.

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- Need 11, not 10!
- This is a common source of errors in programming.
  - "Fencepost errors" or "off-by-one errors".
- Whenever your program involves ranges (1–10, letters "L"–"R"):
  - Test the **boundary cases**.
  - Not just the endpoints, also adjacent numbers.
    - So 0, 1, 2 (lower boundary), 9, 10, 11 (upper).
    - "K", "L", "M" and "Q", "R", "S".
- Why test boundary cases?
  - It's easy to miss one endpoint.
  - Or to go too far, past the endpoint.
  - Make sure in-range inputs are accepted.
  - Make sure out-of-range inputs are rejected.

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what?

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what? Run test 5 again?
    - ★ Make sure you actually fixed it!

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what? Run test 5 again?
    - ★ Make sure you actually fixed it!
- What about tests 1–4?
  - Those tests passed already, right?

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what? Run test 5 again?
    - ★ Make sure you actually fixed it!
- What about tests 1–4?
  - Those tests passed already, right?
  - But what if your change broke something?

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what? Run test 5 again?
    - ⋆ Make sure you actually fixed it!
- What about tests 1–4?
  - Those tests passed already, right?
  - But what if your change broke something?
- A **regression** is when something used to work, but doesn't anymore.
  - Because you (or Python, the OS, . . .) changed something.
  - How to avoid regressions?

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what? Run test 5 again?
    - ⋆ Make sure you actually fixed it!
- What about tests 1–4?
  - Those tests passed already, right?
  - But what if your change broke something?
- A **regression** is when something used to work, but doesn't anymore.
  - Because you (or Python, the OS, . . .) changed something.
  - How to avoid regressions?
- **Regression testing**: whenever you change the code, go back to the beginning and **repeat all the tests**.
  - To make sure you didn't just *add* a bug too!
  - This will save you points on CS 115 programs!

Regression testing

What happens when you find a bug?

- When running your tests, you find an error on test 5.
  - So you fix the bug in your program.
  - Now what? Run test 5 again?
    - ⋆ Make sure you actually fixed it!
- What about tests 1–4?
  - Those tests passed already, right?
  - But what if your change broke something?
- A **regression** is when something used to work, but doesn't anymore.
  - Because you (or Python, the OS, . . .) changed something.
  - How to avoid regressions?
- **Regression testing**: whenever you change the code, go back to the beginning and **repeat all the tests**.
  - To make sure you didn't just *add* a bug too!
  - This will save you points on CS 115 programs!

The end

Next time:

- Libraries and the `math` library.
- Putting it all together: writing a complete program.

The end

Next time:

- Libraries and the `math` library.
- Putting it all together: writing a complete program.