

CS 115 Lecture 3

A first look at Python

Neil Moore

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

3 September 2015

Getting Python and WingIDE

Instructions for installing Python and WingIDE 101 are on the web page:
`http://www.cs.uky.edu/~keen/help/installingpython.html`

We'll use WingIDE today.

Changing the font in WingIDE

Use a big font (18 or 20 point) for labs! It's easier for both us and your teammates.

Changing the font in WingIDE

Use a big font (18 or 20 point) for labs! It's easier for both us and your teammates.

- **Edit** → **Preferences**
- Under “User Interface”, select “Fonts”
 - ▶ May be in a slightly different location on Mac OS.
- Next to “Display Font/Size”:
 - ▶ “Use selected”, then “Change”.
 - ▶ Select a size and click “OK”.

Changing the font in WingIDE

Use a big font (18 or 20 point) for labs! It's easier for both us and your teammates.

- **Edit** → **Preferences**
- Under “User Interface”, select “Fonts”
 - ▶ May be in a slightly different location on Mac OS.
- Next to “Display Font/Size”:
 - ▶ “Use selected”, then “Change”.
 - ▶ Select a size and click “OK”.
- For “Editor Font/Size” (controls your code's font):
 - ▶ Either do the same...
 - ▶ Or select “Match Display Font/Size”

Changing the font in WingIDE

Use a big font (18 or 20 point) for labs! It's easier for both us and your teammates.

- **Edit** → **Preferences**
- Under “User Interface”, select “Fonts”
 - ▶ May be in a slightly different location on Mac OS.
- Next to “Display Font/Size”:
 - ▶ “Use selected”, then “Change”.
 - ▶ Select a size and click “OK”.
- For “Editor Font/Size” (controls your code's font):
 - ▶ Either do the same...
 - ▶ Or select “Match Display Font/Size”
 - ▶ Many people prefer a **monospace** font for code.
 - ★ Consolas, Lucida Console, Courier New, ...

Changing the font in WingIDE

Use a big font (18 or 20 point) for labs! It's easier for both us and your teammates.

- **Edit** → **Preferences**
- Under “User Interface”, select “Fonts”
 - ▶ May be in a slightly different location on Mac OS.
- Next to “Display Font/Size”:
 - ▶ “Use selected”, then “Change”.
 - ▶ Select a size and click “OK”.
- For “Editor Font/Size” (controls your code's font):
 - ▶ Either do the same...
 - ▶ Or select “Match Display Font/Size”
 - ▶ Many people prefer a **monospace** font for code.
 - ★ Consolas, Lucida Console, Courier New, ...

A first Python program, with bugs

```
# Compute the greatest common divisor (GCD) of two numbers.
def main():
    # Inputs: two positive integers (whole numbers) a and b.
    a = input("Please enter a first number: ")
    b = input("Please enter another number: ")
    # 1. Repeat as long as b is not zero:
    while b != 0:
        # 1.1. If a > b, then set a <- (a - b)
        if a > b:
            a = a - b
        # 1.2. Otherwise, set b <- (b - a)
        else:
            b = b - b
    # 2. Output a as the answer.
    print("The GCM of your numbers is", a)
main()
```


Structure of a Python program

- `def main():`
 - ▶ This is the “main function” where the program does all its work
 - ★ (for now)
 - ▶ More about functions in chapter 5.
 - ▶ Python doesn't *need* a main function, but use one in this class!
 - ★ (It's good practice for later.)

Structure of a Python program

- `def main():`
 - ▶ This is the “main function” where the program does all its work
 - ★ (for now)
 - ▶ More about functions in chapter 5.
 - ▶ Python doesn't *need* a main function, but use one in this class!
 - ★ (It's good practice for later.)
- Indentation and blocks.
 - ▶ Code is arranged in indented blocks.
 - ▶ The **body** of main is one block.
 - ▶ It has several blocks inside.

Structure of a Python program

- `def main():`
 - ▶ This is the “main function” where the program does all its work
 - ★ (for now)
 - ▶ More about functions in chapter 5.
 - ▶ Python doesn't *need* a main function, but use one in this class!
 - ★ (It's good practice for later.)
- Indentation and blocks.
 - ▶ Code is arranged in indented blocks.
 - ▶ The **body** of main is one block.
 - ▶ It has several blocks inside.
- `main()`
 - ▶ Calls the main function.
 - ▶ **Not** inside the main function.
 - ★ The `main()` is not indented at all!

Structure of a Python program

- `def main():`
 - ▶ This is the “main function” where the program does all its work
 - ★ (for now)
 - ▶ More about functions in chapter 5.
 - ▶ Python doesn't *need* a main function, but use one in this class!
 - ★ (It's good practice for later.)
- Indentation and blocks.
 - ▶ Code is arranged in indented blocks.
 - ▶ The **body** of `main` is one block.
 - ▶ It has several blocks inside.
- `main()`
 - ▶ Calls the main function.
 - ▶ **Not** inside the main function.
 - ★ The `main()` is not indented at all!
 - ▶ If you forget this line, the program doesn't do anything!

Structure of a Python program

- `def main():`
 - ▶ This is the “main function” where the program does all its work
 - ★ (for now)
 - ▶ More about functions in chapter 5.
 - ▶ Python doesn't *need* a main function, but use one in this class!
 - ★ (It's good practice for later.)
- Indentation and blocks.
 - ▶ Code is arranged in indented blocks.
 - ▶ The **body** of `main` is one block.
 - ▶ It has several blocks inside.
- `main()`
 - ▶ Calls the main function.
 - ▶ **Not** inside the main function.
 - ★ The `main()` is not indented at all!
 - ▶ If you forget this line, the program doesn't do anything!

Documentation and comments

- Syntax: Comments in Python start with a # character and extend to the end of the line.
 - ▶ A variant of comments starts and ends with three single quotes.
 - ▶ This version can include multiple lines, paragraphs, pages.
- Semantics: Does nothing: ignored by Python entirely.
- Why would we want to do that?

Documentation and comments

- Syntax: Comments in Python start with a # character and extend to the end of the line.
 - ▶ A variant of comments starts and ends with three single quotes.
 - ▶ This version can include multiple lines, paragraphs, pages.
- Semantics: Does nothing: ignored by Python entirely.
- Why would we want to do that?
- Comments are for *humans*, not the computer.

Documentation and comments

- Syntax: Comments in Python start with a # character and extend to the end of the line.
 - ▶ A variant of comments starts and ends with three single quotes.
 - ▶ This version can include multiple lines, paragraphs, pages.
- Semantics: Does nothing: ignored by Python entirely.
- Why would we want to do that?
- Comments are for *humans*, not the computer.
 - ▶ Teammates.
 - ▶ Your boss (or instructor, grader, ...)
 - ★ You can talk to your grader while they are grading it!

Documentation and comments

- Syntax: Comments in Python start with a # character and extend to the end of the line.
 - ▶ A variant of comments starts and ends with three single quotes.
 - ▶ This version can include multiple lines, paragraphs, pages.
- Semantics: Does nothing: ignored by Python entirely.
- Why would we want to do that?
- Comments are for *humans*, not the computer.
 - ▶ Teammates.
 - ▶ Your boss (or instructor, grader, . . .)
 - ★ You can talk to your grader while they are grading it!
 - ▶ Yourself next week!

Documentation and comments

- Syntax: Comments in Python start with a # character and extend to the end of the line.
 - ▶ A variant of comments starts and ends with three single quotes.
 - ▶ This version can include multiple lines, paragraphs, pages.
- Semantics: Does nothing: ignored by Python entirely.
- Why would we want to do that?
- Comments are for *humans*, not the computer.
 - ▶ Teammates.
 - ▶ Your boss (or instructor, grader, . . .)
 - ★ You can talk to your grader while they are grading it!
 - ▶ Yourself next week!

Where to use comments

- Comments don't usually need to say *how* you are doing something or *what* you are doing.
 - ▶ That's what the code is for.
- Instead, they should say *why*

Where to use comments

- Comments don't usually need to say *how* you are doing something or *what* you are doing.
 - ▶ That's what the code is for.
- Instead, they should say *why*:
 - BAD: `counter = 0 # set variable to zero`
 - GOOD: `counter = 0 # initialize number of lines`

Where to use comments

- Comments don't usually need to say *how* you are doing something or *what* you are doing.
 - ▶ That's what the code is for.
- Instead, they should say *why*:
 - BAD: `counter = 0 # set variable to zero`
 - GOOD: `counter = 0 # initialize number of lines`
- If the comment is long, put it on a line of its own before the statement.
 - ▶ That way you don't have to scroll horizontally to read it.

Where to use comments

- Comments don't usually need to say *how* you are doing something or *what* you are doing.
 - ▶ That's what the code is for.
- Instead, they should say *why*:
 - BAD: `counter = 0 # set variable to zero`
 - GOOD: `counter = 0 # initialize number of lines`
- If the comment is long, put it on a line of its own before the statement.
 - ▶ That way you don't have to scroll horizontally to read it.
 - ★ In general, try to keep code lines to < 80 characters.
 - ★ Less on team labs, where you should use big fonts.

Where to use comments

- Comments don't usually need to say *how* you are doing something or *what* you are doing.
 - ▶ That's what the code is for.
- Instead, they should say *why*:
 - BAD: `counter = 0 # set variable to zero`
 - GOOD: `counter = 0 # initialize number of lines`
- If the comment is long, put it on a line of its own before the statement.
 - ▶ That way you don't have to scroll horizontally to read it.
 - ★ In general, try to keep code lines to < 80 characters.
 - ★ Less on team labs, where you should use big fonts.

Header comments

All your programming assignments should have a **header comment** at the top.

- See the “Programming Standard” page under “Program Assignments”.
- Doesn't hurt to have them in lab assignments either!

Header comments

All your programming assignments should have a **header comment** at the top.

- See the “Programming Standard” page under “Program Assignments” .
- Doesn't hurt to have them in lab assignments either!
- Name, email, section number
- Purpose of program
- Date completed
- Preconditions: inputs to the program
 - ▶ And what you assume about the inputs.
- Postconditions: outputs of the program.
 - ▶ And what you guarantee about the outputs.

Kinds of errors

Back to our program. . . it has several errors right now.

- Syntax errors
- Semantic errors
- Run-time errors

Syntax errors

- These are the easiest kind to find and fix.
- Syntax is the rules that say how to write statements in the language.

Syntax errors

- These are the easiest kind to find and fix.
- Syntax is the rules that say how to write statements in the language.
 - ▶ Programming languages are very rigid about syntax rules.
 - ▶ Misspelling, wrong punctuation, bad grammar, etc.
 - ▶ Humans can figure out what you meant: not computers.

Syntax errors

- These are the easiest kind to find and fix.
- Syntax is the rules that say how to write statements in the language.
 - ▶ Programming languages are very rigid about syntax rules.
 - ▶ Misspelling, wrong punctuation, bad grammar, etc.
 - ▶ Humans can figure out what you meant: not computers.
- The interpreter (or compiler) will give you an error message.

Syntax errors

- These are the easiest kind to find and fix.
- Syntax is the rules that say how to write statements in the language.
 - ▶ Programming languages are very rigid about syntax rules.
 - ▶ Misspelling, wrong punctuation, bad grammar, etc.
 - ▶ Humans can figure out what you meant: not computers.
- The interpreter (or compiler) will give you an error message.

Semantic errors

- Also known as **logic errors**.
- Semantics = meaning.
 - ▶ The program doesn't do what you wanted it to do.

Semantic errors

- Also known as **logic errors**.
- Semantics = meaning.
 - ▶ The program doesn't do what you wanted it to do.
 - ▶ Maybe you multiplied instead of dividing.
 - ▶ ...or used the wrong variable.

Semantic errors

- Also known as **logic errors**.
- Semantics = meaning.
 - ▶ The program doesn't do what you wanted it to do.
 - ▶ Maybe you multiplied instead of dividing.
 - ▶ ...or used the wrong variable.
- The interpreter **won't** detect these for you!
- So how do we find them?

Semantic errors

- Also known as **logic errors**.
- Semantics = meaning.
 - ▶ The program doesn't do what you wanted it to do.
 - ▶ Maybe you multiplied instead of dividing.
 - ▶ ...or used the wrong variable.
- The interpreter **won't** detect these for you!
- So how do we find them? Testing!
 - ▶ Test plan: what to test, provided input, expected output.

Semantic errors

- Also known as **logic errors**.
- Semantics = meaning.
 - ▶ The program doesn't do what you wanted it to do.
 - ▶ Maybe you multiplied instead of dividing.
 - ▶ ...or used the wrong variable.
- The interpreter **won't** detect these for you!
- So how do we find them? Testing!
 - ▶ Test plan: what to test, provided input, expected output.
 - ▶ Coming up with a good set of test cases is one of the hard parts of programming.
 - ▶ The first part of program 1 will be writing a test plan.

Semantic errors

- Also known as **logic errors**.
- Semantics = meaning.
 - ▶ The program doesn't do what you wanted it to do.
 - ▶ Maybe you multiplied instead of dividing.
 - ▶ ...or used the wrong variable.
- The interpreter **won't** detect these for you!
- So how do we find them? Testing!
 - ▶ Test plan: what to test, provided input, expected output.
 - ▶ Coming up with a good set of test cases is one of the hard parts of programming.
 - ▶ The first part of program 1 will be writing a test plan.

Run-time errors

- The program or interpreter encounters a situation it can't handle.
 - ▶ Usually cause the program to halt with an error message.
 - ▶ Not detected until the situation actually happens!

Run-time errors

- The program or interpreter encounters a situation it can't handle.
 - ▶ Usually cause the program to halt with an error message.
 - ▶ Not detected until the situation actually happens!
- Often caused by the environment (operating system):
 - ▶ File not found.
 - ▶ Network connection closed.
 - ▶ Out of memory.

Run-time errors

- The program or interpreter encounters a situation it can't handle.
 - ▶ Usually cause the program to halt with an error message.
 - ▶ Not detected until the situation actually happens!
- Often caused by the environment (operating system):
 - ▶ File not found.
 - ▶ Network connection closed.
 - ▶ Out of memory.
- Sometimes caused by programming errors:
 - ▶ Used a string where a number was expected.
 - ▶ Undefined variable.

Run-time errors

- The program or interpreter encounters a situation it can't handle.
 - ▶ Usually cause the program to halt with an error message.
 - ▶ Not detected until the situation actually happens!
- Often caused by the environment (operating system):
 - ▶ File not found.
 - ▶ Network connection closed.
 - ▶ Out of memory.
- Sometimes caused by programming errors:
 - ▶ Used a string where a number was expected.
 - ▶ Undefined variable.
- It is possible, but tricky, to catch and handle these errors
 - ▶ **Exception handling**: near the end of the semester.

Run-time errors

- The program or interpreter encounters a situation it can't handle.
 - ▶ Usually cause the program to halt with an error message.
 - ▶ Not detected until the situation actually happens!
- Often caused by the environment (operating system):
 - ▶ File not found.
 - ▶ Network connection closed.
 - ▶ Out of memory.
- Sometimes caused by programming errors:
 - ▶ Used a string where a number was expected.
 - ▶ Undefined variable.
- It is possible, but tricky, to catch and handle these errors
 - ▶ **Exception handling**: near the end of the semester.

Fixing bugs

Let's fix the bugs in our program.

Fixing bugs

Let's fix the bugs in our program.

Fixing bugs

Let's fix the bugs in our program.

- Syntax error: missing indentation.
- Run-time error: input is a string, not a number.
- Semantic error: wrong formula for b .
- Semantic error: output message says "GCM".

Fixing bugs

Let's fix the bugs in our program.

- Syntax error: missing indentation.
- Run-time error: input is a string, not a number.
- Semantic error: wrong formula for b .
- Semantic error: output message says "GCM".

Fixed program

```
# Compute the greatest common divisor (GCD) of two numbers.
def main():
    # Inputs: two positive integers (whole numbers) a and b.
    a = int(input("Please enter a first number: "))
    b = int(input("Please enter another number: "))
    # 1. Repeat as long as b is not zero:
    while b != 0:
        # 1.1. If a > b, then set a <- (a - b)
        if a > b:
            a = a - b
        # 1.2. Otherwise, set b <- (b - a)
        else:
            b = b - a
    # 2. Output a as the answer.
    print("The GCD of your numbers is", a)
main()
```

Variables

A **variable** is a “slot” or “location” that refers to a value.

- a and b were variables in our program.

Variables

A **variable** is a “slot” or “location” that refers to a value.

- a and b were variables in our program.
- A value is something like 42 or "Hello".

Variables

A **variable** is a “slot” or “location” that refers to a value.

- a and b were variables in our program.
- A value is something like 42 or "Hello".
- Variables are stored in RAM.
- They refer to different values as the program runs (vary-able)

Variables

A **variable** is a “slot” or “location” that refers to a value.

- a and b were variables in our program.
- A value is something like 42 or "Hello".
- Variables are stored in RAM.
- They refer to different values as the program runs (vary-able)
 - ▶ **Assignment** (the equals sign) makes a variable refer to a new value.

Variables

A **variable** is a “slot” or “location” that refers to a value.

- a and b were variables in our program.
- A value is something like 42 or "Hello".
- Variables are stored in RAM.
- They refer to different values as the program runs (vary-able)
 - ▶ **Assignment** (the equals sign) makes a variable refer to a new value.
- A fundamental building block of (most) programming languages.

Variables

A **variable** is a “slot” or “location” that refers to a value.

- a and b were variables in our program.
- A value is something like 42 or "Hello".
- Variables are stored in RAM.
- They refer to different values as the program runs (vary-able)
 - ▶ **Assignment** (the equals sign) makes a variable refer to a new value.
- A fundamental building block of (most) programming languages.

Properties of a variable

- Name – what to call the variable?
 - ▶ Also called an “identifier”.

Properties of a variable

- Name – what to call the variable?
 - ▶ Also called an “identifier”.
- Value – what is in the variable?
 - ▶ In Python, the value of a variable is an **object**.

Properties of a variable

- Name – what to call the variable?
 - ▶ Also called an “identifier”.
- Value – what is in the variable?
 - ▶ In Python, the value of a variable is an **object**.
- Type — what kind of value?
 - ▶ Integer, string, floating-point number, ...

Properties of a variable

- Name – what to call the variable?
 - ▶ Also called an “identifier”.
- Value – what is in the variable?
 - ▶ In Python, the value of a variable is an **object**.
- Type — what kind of value?
 - ▶ Integer, string, floating-point number, ...
- Scope – where in the program is the name valid?
 - ▶ In Python, goes from the definition to the end of that block.

Properties of a variable

- Name – what to call the variable?
 - ▶ Also called an “identifier”.
- Value – what is in the variable?
 - ▶ In Python, the value of a variable is an **object**.
- Type — what kind of value?
 - ▶ Integer, string, floating-point number, ...
- Scope – where in the program is the name valid?
 - ▶ In Python, goes from the definition to the end of that block.
 - ▶ Can have variables with the same name as long as their scopes don't overlap.
 - ★ They're entirely unrelated variables!

Properties of a variable

- Name – what to call the variable?
 - ▶ Also called an “identifier”.
- Value – what is in the variable?
 - ▶ In Python, the value of a variable is an **object**.
- Type — what kind of value?
 - ▶ Integer, string, floating-point number, ...
- Scope – where in the program is the name valid?
 - ▶ In Python, goes from the definition to the end of that block.
 - ▶ Can have variables with the same name as long as their scopes don't overlap.
 - ★ They're entirely unrelated variables!

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: `students` and `Students` and `STUDENTS` are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: students and Students and STUDENTS are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).
 - ▶ Cannot be a **reserved word** (if, while, etc.)

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: students and Students and STUDENTS are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).
 - ▶ Cannot be a **reserved word** (if, while, etc.)
 - ★ Dark blue in WingIDE.
- OK: x, size, name2, long_name, CamelCase, _ugly

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: students and Students and STUDENTS are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).
 - ▶ Cannot be a **reserved word** (if, while, etc.)
 - ★ Dark blue in WingIDE.
- OK: x, size, name2, long_name, CamelCase, _ugly
- **BAD**: 2bad4u, no spaces, no-punctuation

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: students and Students and STUDENTS are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).
 - ▶ Cannot be a **reserved word** (if, while, etc.)
 - ★ Dark blue in WingIDE.
- OK: x, size, name2, long_name, CamelCase, _ugly
- **BAD: 2bad4u, no spaces, no-punctuation**
- Just because it’s legal doesn’t mean it’s good.
 - ▶ Avoid single-letter variables.
 - ★ Except in loop counters, simple math functions.
 - ★ thing, number aren’t any better.

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: students and Students and STUDENTS are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).
 - ▶ Cannot be a **reserved word** (if, while, etc.)
 - ★ Dark blue in WingIDE.
- OK: x, size, name2, long_name, CamelCase, _ugly
- **BAD: 2bad4u, no spaces, no-punctuation**
- Just because it’s legal doesn’t mean it’s good.
 - ▶ Avoid single-letter variables.
 - ★ Except in loop counters, simple math functions.
 - ★ thing, number aren’t any better.
 - ▶ Instead of n, perhaps count

Identifiers (variable names)

- A sequence of letters, digits, and underscores
 - ▶ “Alphanumeric” characters.
 - ▶ **Case sensitive**: students and Students and STUDENTS are all different.
 - ▶ Cannot start with a digit (Python thinks that’s a number).
 - ▶ Cannot be a **reserved word** (if, while, etc.)
 - ★ Dark blue in WingIDE.
- OK: x, size, name2, long_name, CamelCase, _ugly
- **BAD**: 2bad4u, no spaces, no-punctuation
- Just because it’s legal doesn’t mean it’s good.
 - ▶ Avoid single-letter variables.
 - ★ Except in loop counters, simple math functions.
 - ★ thing, number aren’t any better.
 - ▶ Instead of n, perhaps count
 - ★ Even better: num_students

Can variable properties change?

- The name and scope of a variable never change.
 - ▶ If it looks like it did: it's a different variable.

Can variable properties change?

- The name and scope of a variable never change.
 - ▶ If it looks like it did: it's a different variable.
- In a “dynamically typed” language like Python, the value and type of a variable can change.
 - ▶ With an assignment statement:
`score = 0.0`
`score = "incomplete"`

Can variable properties change?

- The name and scope of a variable never change.
 - ▶ If it looks like it did: it's a different variable.
- In a “dynamically typed” language like Python, the value and type of a variable can change.
 - ▶ With an assignment statement:
`score = 0.0`
`score = "incomplete"`
- In “statically typed” languages like C++, the type cannot change.
 - ▶ Even in Python, it's less confusing if each variable has one type.

Can variable properties change?

- The name and scope of a variable never change.
 - ▶ If it looks like it did: it's a different variable.
- In a “dynamically typed” language like Python, the value and type of a variable can change.
 - ▶ With an assignment statement:
`score = 0.0`
`score = "incomplete"`
- In “statically typed” languages like C++, the type cannot change.
 - ▶ Even in Python, it's less confusing if each variable has one type.
 - ▶ One common style: include the type in the variable name:
 - ★ `user_list, name_str, ...`

Can variable properties change?

- The name and scope of a variable never change.
 - ▶ If it looks like it did: it's a different variable.
- In a “dynamically typed” language like Python, the value and type of a variable can change.
 - ▶ With an assignment statement:

```
score = 0.0
score = "incomplete"
```
- In “statically typed” languages like C++, the type cannot change.
 - ▶ Even in Python, it's less confusing if each variable has one type.
 - ▶ One common style: include the type in the variable name:
 - ★ `user_list, name_str, ...`
- In “pure functional” languages like Haskell, the *value* cannot change!
 - ▶ So maybe “**variable**” is not the right word there!

Can variable properties change?

- The name and scope of a variable never change.
 - ▶ If it looks like it did: it's a different variable.
- In a “dynamically typed” language like Python, the value and type of a variable can change.
 - ▶ With an assignment statement:

```
score = 0.0  
score = "incomplete"
```
- In “statically typed” languages like C++, the type cannot change.
 - ▶ Even in Python, it's less confusing if each variable has one type.
 - ▶ One common style: include the type in the variable name:
 - ★ `user_list, name_str, ...`
- In “pure functional” languages like Haskell, the *value* cannot change!
 - ▶ So maybe “**variable**” is not the right word there!

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means "add one to x ".

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means “add one to x ”.
 - ▶ Maybe better to pronounce it “gets” than “equals”.

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means “add one to x ”.
 - ▶ Maybe better to pronounce it “gets” than “equals”.
 - ▶ “Assign $x + 1$ to x ” or “Assign x with/from $x + 1$ ”.

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means “add one to x ”.
 - ▶ Maybe better to pronounce it “gets” than “equals”.
 - ▶ “Assign $x + 1$ to x ” or “Assign x with/from $x + 1$ ”.
- Order matters!
 - ▶ Performs the calculation on the *right*.
 - ▶ Changes *only* the variable on the *left*.

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means “add one to x ”.
 - ▶ Maybe better to pronounce it “gets” than “equals”.
 - ▶ “Assign $x + 1$ to x ” or “Assign x with/from $x + 1$ ”.
- Order matters!
 - ▶ Performs the calculation on the *right*.
 - ▶ Changes *only* the variable on the *left*.
 - ▶ $x + 1 = x$ # Syntax error!

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means “add one to x ”.
 - ▶ Maybe better to pronounce it “gets” than “equals”.
 - ▶ “Assign $x + 1$ to x ” or “Assign x with/from $x + 1$ ”.
- Order matters!
 - ▶ Performs the calculation on the *right*.
 - ▶ Changes *only* the variable on the *left*.
 - ▶ $x + 1 = x$ # Syntax error!
- If the LHS variable doesn't already exist in this scope, creates it.
 - ▶ “Initialization”: giving a variable its initial value.

Assignment

- Syntax: *variable = expression*
- Semantics:
Calculates the value of (**evaluates**) the right hand side (RHS), then changes the value of the variable on the left hand side (LHS).
 - ▶ In a later class we'll see other things that can go on the LHS.
- Not an equation!
 - ▶ In math, $x = x + 1$ has no solution.
 - ▶ But in Python, $x = x + 1$ means “add one to x ”.
 - ▶ Maybe better to pronounce it “gets” than “equals”.
 - ▶ “Assign $x + 1$ to x ” or “Assign x with/from $x + 1$ ”.
- Order matters!
 - ▶ Performs the calculation on the *right*.
 - ▶ Changes *only* the variable on the *left*.
 - ▶ $x + 1 = x$ # Syntax error!
- If the LHS variable doesn't already exist in this scope, creates it.
 - ▶ “Initialization”: giving a variable its initial value.

Using assignment: swapping

Suppose we have two variables and want to swap their values.

- So each variable's new value is the other variable's old value

Using assignment: swapping

Suppose we have two variables and want to swap their values.

- So each variable's new value is the other variable's old value:

```
x = 10
```

```
y = 42
```

```
# do something
```

```
print(x, y) # should print: 42 10
```

Using assignment: swapping

Suppose we have two variables and want to swap their values.

- So each variable's new value is the other variable's old value:

```
x = 10
```

```
y = 42
```

```
# do something
```

```
print(x, y) # should print: 42 10
```

- Will this work?

```
x = y
```

```
y = x
```

```
print(x, y)
```

Using assignment: swapping

Suppose we have two variables and want to swap their values.

- So each variable's new value is the other variable's old value:

```
x = 10
```

```
y = 42
```

```
# do something
```

```
print(x, y) # should print: 42 10
```

- Will this work?

```
x = y
```

```
y = x
```

```
print(x, y) ⇒ 42 42
```

- We lost the old value of x!

Using assignment: swapping

Suppose we have two variables and want to swap their values.

- So each variable's new value is the other variable's old value:

```
x = 10
y = 42
# do something
print(x, y) # should print: 42 10
```

- Will this work?

```
x = y
y = x
print(x, y) ⇒ 42 42
```

- We lost the old value of x! Need a temporary variable:

```
temp = x
x = y
y = temp
```

Using assignment: swapping

Suppose we have two variables and want to swap their values.

- So each variable's new value is the other variable's old value:

```
x = 10
y = 42
# do something
print(x, y) # should print: 42 10
```

- Will this work?

```
x = y
y = x
print(x, y) ⇒ 42 42
```

- We lost the old value of x! Need a temporary variable:

```
temp = x
x = y
y = temp
```

Simple arithmetic

- The **expression** on the right hand side can be an arithmetic expression.
- Arithmetic operators in Python are:
 - ▶ +, - (add and subtract: $a + b$, $c - d$)
 - ▶ * (multiply), / (divide)
 - ▶ ** (exponentiate, “to the”)

Simple arithmetic

- The **expression** on the right hand side can be an arithmetic expression.
- Arithmetic operators in Python are:
 - ▶ +, - (add and subtract: $a + b$, $c - d$)
 - ▶ * (multiply), / (divide)
 - ▶ ** (exponentiate, “to the”)
- Order of operations:
 - ▶ ** first (highest **precedence**)
 - ▶ Then * and /
 - ▶ Then + and - (lowest **precedence**)

Simple arithmetic

- The **expression** on the right hand side can be an arithmetic expression.
- Arithmetic operators in Python are:
 - ▶ +, - (add and subtract: $a + b$, $c - d$)
 - ▶ * (multiply), / (divide)
 - ▶ ** (exponentiate, “to the”)
- Order of operations:
 - ▶ ** first (highest **precedence**)
 - ▶ Then * and /
 - ▶ Then + and - (lowest **precedence**)
 - ▶ Can use parentheses to make the order explicit:
`total = price * (tax + 100) / 100`

Simple arithmetic

- The **expression** on the right hand side can be an arithmetic expression.
- Arithmetic operators in Python are:
 - ▶ +, - (add and subtract: $a + b$, $c - d$)
 - ▶ * (multiply), / (divide)
 - ▶ ** (exponentiate, “to the”)
- Order of operations:
 - ▶ ** first (highest **precedence**)
 - ▶ Then * and /
 - ▶ Then + and - (lowest **precedence**)
 - ▶ Can use parentheses to make the order explicit:
`total = price * (tax + 100) / 100`
- We'll see more details about these operators next time when we talk about types.

Simple arithmetic

- The **expression** on the right hand side can be an arithmetic expression.
- Arithmetic operators in Python are:
 - ▶ +, - (add and subtract: $a + b$, $c - d$)
 - ▶ * (multiply), / (divide)
 - ▶ ** (exponentiate, “to the”)
- Order of operations:
 - ▶ ** first (highest **precedence**)
 - ▶ Then * and /
 - ▶ Then + and - (lowest **precedence**)
 - ▶ Can use parentheses to make the order explicit:
`total = price * (tax + 100) / 100`
- We'll see more details about these operators next time when we talk about types.

Next time

- Data types in Python.
- More about arithmetic.
- Getting input.
- Printing.
- Testing.