Lecture 2: Variables, types, and operators

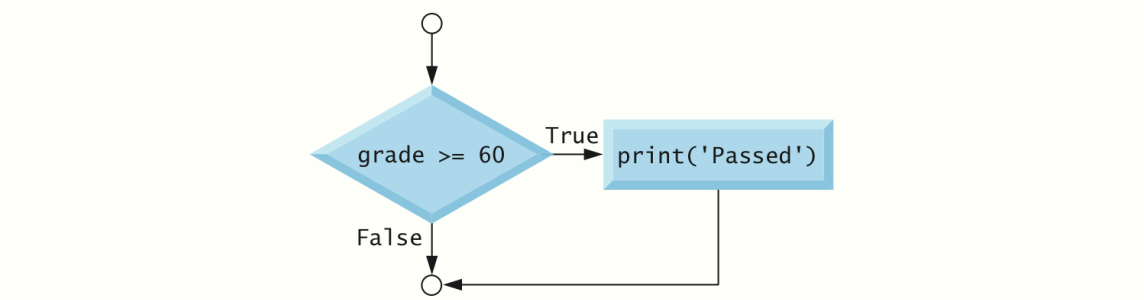
INF 605 · 2026-08-27

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. Last class was **Introduction to Computers and Python**. Name one thing we actually ran or wrote.
2. Last class, which name or API should you still remember for today's first demo?



Textbook flowchart. True takes one path; False takes the other.

2.2 Variables and Assignment Statements (1 of 2)

```
In [5]: 45 + 72
```

```
Out[5]: 117
```

The next cell continues the same idea — read the new names before you run it.

```
In [7]: x = 7
```

2.2 Variables and Assignment Statements (2 of 2)

- Preceding snippet is a **statement**.
- Specifies a task to perform.
- Preceding statement creates `x` and uses the **assignment symbol (`=`)** to give `x` a value.

In [9]:
```
y = 3
```

Adding Variable Values and Viewing the Result

In [11]:
```
# Note: if you don't run previous cells, you will get an error message

x + y
```

Out[11]:
```
10
```

Calculations in Assignment Statements

In [13]:
```
total = x + y
```

- **assignment symbol (`=`)** is not an operator

In [15]:
```
total
```

Out[15]:
```
10
```

Python Style

- The *Style Guide for Python Code* helps you write code that conforms to Python's coding conventions.
- Recommends inserting one space on each side of the assignment symbol `=` and binary operators like `+` to make programs more readable.

Variable Names

- A variable name is an **identifier**.
- May consist of letters, digits and underscores (`_`) but may not begin with a digit.
- Python is *case sensitive*.

In [18]:
```
x = 7
```

Types (1 of 2)

- Each value in Python has a type that indicates the kind of data the value represents.
- You can view a value's type, with the **`type` built-in function**.

In [20]:
```
type()
```

```
Traceback (most recent call last):
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classr
oom_nb.py", line 271, in execute_notebook
    last_val, had_expr = _exec_with_result(runnable, env, f"cell_{i}")
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classr
oom_nb.py", line 167, in _exec_with_result
    last_val = eval(compile(ast.Expression(tree.body[-1].value), filename,
"eval"), env)
  File "cell_19", line 1, in <module>
TypeError: type() takes 1 or 3 arguments
```

The next cell continues the same idea — read the new names before you run it.

In [22]:
```
type(10.5)
```

Out[22]:
```
<class 'float'>
```

Types (2 of 2)

- A function performs a task when you call it by writing its name, followed by **parentheses,** `()`.
- The parentheses contain the function's **argument**—the data that the type function needs to perform its task.

2.3 Arithmetic

Python operation	Arithmetic operator	Python expression
Addition	`+`	`f + 7`
Subtraction	`-`	`p - c`
Multiplication	`*`	`b * m`
Exponentiation	`**`	`x ** y`

Python operation	Arithmetic operator	Python expression
True division	`/`	`x / y`
Floor division	`//`	`x // y`
Remainder (modulo)	`%`	`r % s`

- All operators and their precedence

Multiplication (`*`)

- Python uses the **asterisk (`*`) multiplication operator**:

```
In [27]: 7 * 4
```

```
Out[27]: 28
```

Exponentiation (`**`)

- The **exponentiation (**) operator** raises one value to the power of another.

```
In [29]: 2 ** 10
```

```
Out[29]: 1024
```

- To calculate the square root, use the exponent `1/2` or `0.5` .

```
In [31]: 9 ** (1 / 2)
```

```
Out[31]: 3.0
```

The next cell continues the same idea — read the new names before you run it.

```
In [33]: 9 ** (0.5)
```

```
Out[33]: 3.0
```

True Division (`/`) vs. Floor Division (`//`) (1 of 3)

- **True division (`/`)** divides a numerator by a denominator and yields a floating-point number.

```
In [35]: 7 / 4
```

```
Out[35]: 1.75
```

True Division (`/`) vs. Floor Division (`//`) (2 of 3)

- **Floor division (`//`)** divides a numerator by a denominator, yielding the highest *integer* that's not greater than the result.
- **Truncates** (discards) the fractional part.

In [37]:
```
7 // 4
```

Out[37]: 1

The next cell continues the same idea — read the new names before you run it.

In [39]:
```
3 / 5
```

Out[39]: 0.6

The next cell continues the same idea — read the new names before you run it.

In [41]:
```
3 // 5
```

Out[41]: 0

The next cell continues the same idea — read the new names before you run it.

In [43]:
```
type(14 // 7)
```

Out[43]: <class 'int'>

True Division (`/`) vs. Floor Division (`//`) (3 of 3)

In [45]:
```
-13 / 4
```

Out[45]: -3.25

The next cell continues the same idea — read the new names before you run it.

In [47]:
```
-13 // 4
```

Out[47]: -4

The next cell continues the same idea — read the new names before you run it.

In [49]:
```
2 * 3
```

Out[49]: 6

Exceptions and Tracebacks (1 of 3)

- Dividing by zero with `/` or `//` is not allowed and results in an **exception**.

In [51]:
```
123 / 0
```

```
Traceback (most recent call last):
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classr
oom_nb.py", line 271, in execute_notebook
    last_val, had_expr = _exec_with_result(runnable, env, f"cell_{i}")
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classr
oom_nb.py", line 167, in _exec_with_result
    last_val = eval(compile(ast.Expression(tree.body[-1].value), filename,
"eval"), env)
  File "cell_50", line 1, in <module>
ZeroDivisionError: division by zero
```

Exceptions and Tracebacks (2 of 3)

- Exceptions produce **tracebacks**.
- The line that begins with `---->` shows the code that caused the exception.
- The error message at the bottom of the traceback shows the exception that occurred, followed by a colon (:) and an error message with more information about the exception..

Exceptions and Tracebacks (3 of 3)

- An exception occurs if you try to use a variable that you have not yet created.

In [54]:
```
z + 7
```

Remainder Operator

- **Remainder operator (`%`)** yields the remainder after the left operand is divided by the right operand.

In [56]:
```
17 % 5
```

Out[56]: 2

The next cell continues the same idea — read the new names before you run it.

In [58]:
```
7.5 % 3.5
```

Out[58]: 0.5

Straight-Line Form

- Algebraic expressions must be typed in **straight-line form** using Python's operators.

Grouping Expressions with Parentheses

- Parentheses group Python expressions, as in algebraic expressions.

```
In [61]: 10 * (5 + 3)
```

```
Out[61]: 80
```

The next cell continues the same idea — read the new names before you run it.

```
In [63]: 10 * 5 + 3
```

```
Out[63]: 53
```

Operator Precedence Rules (1 of 2)

- Generally the same as those in algebra:

> 1. Expressions in parentheses evaluate first, so parentheses may force the order of evaluation to occur in any sequence you desire. Parentheses have the highest level of precedence. In expressions with **nested parentheses**, such as `(a / (b - c))`, the expression in the *innermost* parentheses (that is, `b - c`) evaluates first.
> 2. Exponentiation operations evaluate next. If an expression contains several exponentiation operations, Python applies them from right to left.

Operator Precedence Rules (1 of 2)

> 3. Multiplication, division and modulus operations evaluate next. If an expression contains several multiplication, true-division, floor-division and modulus operations, Python applies them from left to right. Multiplication, division and modulus are "on the same level of precedence."
> 4. Addition and subtraction operations evaluate last. If an expression contains several addition and subtraction operations, Python applies them from left to right. Addition and subtraction also have the same level of precedence.

- Complete list of operators and their precedence

Operator Grouping

- When we say that Python applies certain operators from left to right, we are referring to the operators' **grouping**.
- All Python operators of the same precedence group left-to-right except for the exponentiation operator (`**`), which groups right-to-left.

Redundant Parentheses

- Can use redundant parentheses to group subexpressions to make an expression clearer.

Operand Types

- If both operands are integers, the result is an integer—**except for the true-division (`/`) operator, which always yields a floating-point number**.
- If both operands are floating-point numbers, the result is a floating-point number.
- Mixed-type expressions produce floating-point results.

2.4 Function `print` and an Intro to Single- and-Double-Quoted Strings

- The built-in **`print` function** displays its argument(s) as a line of text

In [71]:
```
print('Welcome to Python!')
```

```
Welcome to Python!
```

- May enclose a string in double quotes (`"`).

In [73]:
```
print("Welcome to Python!")
```

```
Welcome to Python!
```

- Python programmers generally prefer single quotes.
- When `print` completes its task, it positions the screen cursor at the beginning of the next line.

In [75]:
```
print('Hello')
print('World')
```

```
Hello
World
```

Printing a Comma-Separated List of Items

In [77]:
```
print('Welcome', 'to', 'Python!')
```

```
Welcome to Python!
```

- Displays each argument separated from the next by a space.

Printing Many Lines of Text with One Statement

- A backslash (`\`) in a string is the **escape character**.
- The backslash and the character immediately following it form an **escape sequence**.
- `\n` represents the **newline character** escape sequence, which tells `print` to move the output cursor to the next line.

In [80]:
```
print('Welcome\nto\n\nPython!')
```

```
Welcome
to

Python!
```

Other Escape Sequences

Escape sequence	Description
`\n`	Insert a newline character in a string. When the string is displayed, for each newline, move the screen cursor to the beginning of the next line.
`\t`	Insert a horizontal tab. When the string is displayed, for each tab, move the screen cursor to the next tab stop.

Escape sequence	Description
`\\`	Insert a backslash character in a string.
`\"`	Insert a double quote character in a string.
`\'`	Insert a single quote character in a string.

Ignoring a Line Break in a Long String

- Can split a long string (or a long statement) over several lines by using the **\ continuation character** as the last character on a line to ignore the line break.

```
In [83]: print('this is a longer string, so we \
         split it over two lines')
```

```
this is a longer string, so we split it over two lines
```

- In this case, `\` is not the escape character because another character does not follow it.

Printing the Value of an Expression

```
In [86]: print('Sum is', 7 + 3)
```

```
Sum is 10
```

2.5 Triple-Quoted Strings

- Delimited by `"""` or `'''`, but the *Style Guide for Python Code* recommends `"""`.
- Used for:

- multiline strings
- strings containing single or double quotes
- **docstrings**—the recommended way to document the purposes of certain program components.

Including Quotes in Strings (1 of 3)

- A string delimited by single quotes may include double-quote characters, but not single quotes, unless you use the `\'` escape sequence.

In [90]:
```
print('Display "hi" in quotes')
```

```
Display "hi" in quotes
```

The next cell continues the same idea — read the new names before you run it.

In [92]:
```
print('Display 'hi' in quotes')
```

```
Traceback (most recent call last):
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classr
oom_nb.py", line 271, in execute_notebook
    last_val, had_expr = _exec_with_result(runnable, env, f"cell_{i}")
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classr
oom_nb.py", line 159, in _exec_with_result
    tree = ast.parse(src)
  File "/Library/Developer/CommandLineTools/Library/Frameworks/Python3.frame
work/Versions/3.9/lib/python3.9/ast.py", line 50, in parse
    return compile(source, filename, mode, flags,
  File "<unknown>", line 1
    print('Display 'hi' in quotes')
                    ^
SyntaxError: invalid syntax
```

The next cell continues the same idea — read the new names before you run it.

In [94]:
```
print('Display \'hi\' in quotes')
```

```
Display 'hi' in quotes
```

Including Quotes in Strings (2 of 3)

- A string delimited by double quotes may include single quote characters, but not double quotes, unless you use the `\"` escape sequence.

In [96]:
```
print("Display the name O'Brien")
```

```
Display the name O'Brien
```

The next cell continues the same idea — read the new names before you run it.

In [98]:
```
print("Display \"hi\" in quotes")
```

```
Display "hi" in quotes
```

Including Quotes in Strings (3 of 3)

- Triple-quoted strings may contain both single and double quotes.

In [100…
```
print("""Display "hi" and 'bye' in quotes""")
```

```
Display "hi" and 'bye' in quotes
```

Multiline Strings (1 of 2)

In [102…
```
triple_quoted_string = """This is a triple-quoted
string that spans two lines"""
```

Multiline Strings (2 of 2)

In [104…
```
print(triple_quoted_string)
```

```
This is a triple-quoted
string that spans two lines
```

- Python stores multiline strings with embedded newline characters.

In [106…
```
triple_quoted_string
```

Out[106… `'This is a triple-quoted\nstring that spans two lines'`

Exercise (extra-1)

Using a name already defined above, produce one printed result. Keep it short — this checks the section you just read.

In [ ]:
```
# Exercise (extra-1) — your attempt
result = None  # fill using a name from the cells above
print(result)
```

Solution (extra-1).

In [111…
```
# Solution (extra-1)
print('see the demo above')
```

```
see the demo above
```

Comprehensive exercises

Work these if we finish early. They use more than one idea from today.

Exercise (together-1)

Using names already in this notebook, print one result that mixes two ideas from today.

In [ ]:
```
# Exercise (together-1) — your attempt
result = None  # fill using today's names
print(result)
```

Solution (together-1).

In [116…
```
# Solution (together-1)
print('combine two demos from above')
```

```
combine two demos from above
```

Exercise (together-2)

Write a 2-line check that today's main object still has the shape or length you expect.

In [ ]:
```
# Exercise (together-2) — your attempt
ok = None
print(ok)
```

Solution (together-2).

In [120…
```
# Solution (together-2)
print(True)
```

```
True
```

Takeaways

- Today's notebook is the lecture.
- Save a copy in Drive.

- Next meeting continues from here.