2. Introduction to Python Programming

Objectives (1 of 3)

In this chapter, you'll:

- Continue using IPython interactive mode to enter code snippets and see their results immediately.
- Write simple Python statements and scripts.
- Create variables to store data for later use.
- Become familiar with built-in data types.

Objectives (2 of 3)

- Use built-in function `print` to display text.
- Use arithmetic operators and comparison operators, and understand their precedence.
- Use single-, double- and triple-quoted strings.
- Use built-in function `input` to prompt the user to enter data at the keyboard and get that data for use in the program.

Objectives (3 of 3)

- Convert text to integer values with built-in function `int` .
- Use comparison operators and the `if` statement to decide whether to execute a statement or group of statements.
- Learn about objects and Python's dynamic typing.
- Use built-in function `type` to get an object's type.

Outline

Outline (cont.)

2.2 Variables and Assignment Statements

2.2 Variables and Assignment Statements (1 of 2)

In [1]:

```
45 + 72
```

Out[1]:

```
117
```

In [2]:

```
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 [3]:

```
y = 3
```

Adding Variable Values and Viewing the Result

In [4]:

```
x + y
```

Out[4]:

```
10
```

Calculations in Assignment Statements

In [5]:

```
total = x + y
```

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

In [6]:

```
total
```

Out[6]:

```
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*.

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 [7]:

```
type(x)
```

Out[7]:

```
int
```

In [8]:

```
type(10.5)
```

Out[8]:

```
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`
True division	`/`	`x / y`
Floor division	`//`	`x // y`
Remainder (modulo)	`%`	`r % s`

- All operators and their precedence (https://docs.python.org/3/reference/expressions.html#operator-precedence)

Multiplication (*)

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

In [1]:

```
7 * 4
```

Out[1]:

```
28
```

Exponentiation (**)

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

In [2]:

```
2 ** 10
```

Out[2]:

```
1024
```

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

In [3]:

```
9 ** (1 / 2)
```

Out[3]:

```
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 [4]:

```
7 / 4
```

Out[4]:

```
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 [5]:

```
7 // 4
```

Out[5]:

```
1
```

In [6]:

```
3 // 5
```

Out[6]:

```
0
```

In [7]:

```
14 // 7
```

Out[7]:

```
2
```

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

In [8]:

```
-13 / 4
```

Out[8]:

```
-3.25
```

```
In [9]:
```

```
-13 // 4
```

```
Out[9]:

-4
```

Exceptions and Tracebacks (1 of 3)

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

```
In [10]:
```

```
123 / 0
```

```
---------------------------------------------------------------------------
----
ZeroDivisionError                         Traceback (most recent call l
ast)
<ipython-input-10-76e1a9ab9410> in <module>
----> 1 123 / 0

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 [11]:
```

```
z + 7
```

```
---------------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-11-2ca5f6c7aca2> in <module>
----> 1 z + 7

NameError: name 'z' is not defined
```

Remainder Operator

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

In [12]:

```
17 % 5
```

Out[12]:

```
2
```

In [13]:

```
7.5 % 3.5
```

Out[13]:

```
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 [14]:

```
10 * (5 + 3)
```

Out[14]:

```
80
```

In [15]:

```
10 * 5 + 3
```

Out[15]:

```
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)

> 1. 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.”
> 2. 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 (https://docs.python.org/3/reference/expressions.html#operator-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 [1]:

```
print('Welcome to Python!')
```

```
Welcome to Python!
```

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

In [2]:

```
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.

Printing a Comma-Separated List of Items

In [3]:

```
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 [4]:

```
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.
`\\`	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 [5]:

```
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 [6]:

```
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 [1]:

```
print('Display "hi" in quotes')
```

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

In [2]:

```
print('Display 'hi' in quotes')
```

```
  File "<ipython-input-2-19bf596ccf72>", line 1
    print('Display 'hi' in quotes')
                     ^
SyntaxError: invalid syntax
```

In [3]:

```
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 [4]:

```
print("Display the name O'Brien")
```

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

In [5]:

```
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 [6]:
```

```python
print("""Display "hi" and 'bye' in quotes""")
```

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

Multiline Strings (1 of 2)

```
In [7]:
```

```python
triple_quoted_string = """This is a triple-quoted
string that spans two lines"""
```

- IPython knows that the string is incomplete because we did not type the closing `"""` before we pressed *Enter*.
- IPython displays a **continuation prompt** `...:` at which you can input the multiline string's next line.
- This continues until you enter the ending `"""` and press *Enter*.

Multiline Strings (2 of 2)

```
In [8]:
```

```python
print(triple_quoted_string)
```

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

- Python stores multiline strings with embedded newline characters.

```
In [9]:
```

```python
triple_quoted_string
```

```
Out[9]:
```

```
'This is a triple-quoted\nstring that spans two lines'
```

2.6 Getting Input from the User

- Built-in **`input` function** requests and obtains user input.

In [1]:

```
name = input("What's your name? ")
```

In [2]:

```
name
```

Out[2]:

```
'Paul'
```

In [3]:

```
print(name)
```

```
Paul
```

- If you enter quotes, they're input as part of the string.

In [4]:

```
name = input("What's your name? ")
```

In [5]:

```
name
```

Out[5]:

```
'"Paul"'
```

In [6]:

```
print(name)
```

```
"Paul"
```

Function `input` Always Returns a String

In [7]:

```
value1 = input('Enter first number: ')
```

In [8]:

```
value2 = input('Enter second number: ')
```

In [9]:

```
value1 + value2
```

Out[9]:

```
'73'
```

- Python "adds" the *string* values `'7'` and `'3'`, producing the *string* `'73'`.
- Known as **string concatenation.**

Getting an Integer from the User

- If you need an integer, convert the string to an integer using the built-in **`int` function.**

In [10]:

```
value = input('Enter an integer: ')
```

In [11]:

```
value = int(value)
```

In [12]:

```
value
```

Out[12]:

```
7
```

- Can combine `int` and `input` in one statement.

In [13]:

```
another_value = int(input('Enter another integer: '))
```

In [14]:

```
another_value
```

Out[14]:

```
3
```

In [15]:

```
value + another_value
```

Out[15]:

```
10
```

- If the string passed to `int` cannot be converted to an integer, a `ValueError` occurs.

```
In [16]:
```

```
bad_value = int(input('Enter another integer: '))
```

```
---------------------------------------------------------------------------
----
ValueError                                Traceback (most recent call l
ast)
<ipython-input-16-cd36e6cf8911> in <module>
----> 1 bad_value = int(input('Enter another integer: '))

ValueError: invalid literal for int() with base 10: 'hello'
```

- Function `int` also can convert a floating-point value to an integer.

```
In [17]:
```

```
int(10.5)
```

```
Out[17]:

10
```

2.7 Decision Making: The if Statement and Comparison Operators

- A **condition** is a Boolean expression with the value `True` or `False`.

In [1]:

```
7 > 4
```

Out[1]:

```
True
```

In [2]:

```
7 < 4
```

Out[2]:

```
False
```

Algebraic operator	Python operator	Sample condition	Meaning
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`
=	`==`	`x == y`	`x` is equal to `y`
≠	`!=`	`x != y`	`x` is not equal to `y`

- Operators `>`, `<`, `>=` and `<=` have the same precedence.
- Operators `==` and `!=` have the same precedence, which is lower than `>`, `<`, `>=` and `<=`.

- It's a syntax error when any of the operators `==`, `!=`, `>=` and `<=` contains spaces between its pair of symbols.

In [3]:

```
7 > = 4
```

```
  File "<ipython-input-3-5c6e2897f3b3>", line 1
    7 > = 4
        ^
SyntaxError: invalid syntax
```

Making Decisions with the if Statement: Introducing Scripts

- The **`if` statement** uses a condition to decide whether to execute a statement (or a group of statements).
- We'll read two integers from the user and compare them using six consecutive `if` statements, one for each comparison operator.
- When you have many statements to execute as a group, you typically write them as a **script** stored in a file with the `.py` (short for Python) extension.

- To run this example, change to this chapter's `ch02` examples folder, then enter:

```
ipython fig02_01.py
```

- In IPython interactive mode or in JupyterLab, you can use the command:

```
run fig02_01.py
```

```python
# fig02_01.py
"""Comparing integers using if statements and comparison operators."""

print('Enter two integers, and I will tell you',
      'the relationships they satisfy.')

# read first integer
number1 = int(input('Enter first integer: '))

# read second integer
number2 = int(input('Enter second integer: '))

if number1 == number2:
    print(number1, 'is equal to', number2)

if number1 != number2:
    print(number1, 'is not equal to', number2)

if number1 < number2:
    print(number1, 'is less than', number2)

if number1 > number2:
    print(number1, 'is greater than', number2)

if number1 <= number2:
    print(number1, 'is less than or equal to', number2)

if number1 >= number2:
    print(number1, 'is greater than or equal to', number2)
```

- Enter `37` and `42`

In [4]:

```
run fig02_01.py
```

```
Enter two integers and I will tell you the relationships they satisf
y.

37 is not equal to 42
37 is less than 42
37 is less than or equal to 42
```

- Enter `7` and `7`

In [5]:

```
run fig02_01.py
```

```
Enter two integers and I will tell you the relationships they satisf
y.


7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

- Enter `54` and `17`

In [6]:

```
run fig02_01.py
```

```
Enter two integers and I will tell you the relationships they satisf
y.

54 is not equal to 17
54 is greater than 17
54 is greater than or equal to 17
```

sqqi rxw

- The hash character (`#`) indicates that the rest of the line is a **comment**.
- We begin each script with a comment indicating the script’s file name.
- A comment also can begin to the right of the code on a given line and continue until the end of that line.

sgwxmr w

- The *Style Guide for Python Code* says each script should start with a docstring that explains the script’s purpose.
- Often spans many lines for more complex scripts.

Fpero mriw

- Blank lines and space characters to make code easier to read.
- Together, blank lines, space characters and tab characters are known as **white space**.
- Python ignores most white space.

tpxmr e ir xl xexiqi rxEgvsww mriw

- Typically, you write statements on one line.
- You may spread a lengthy statement over several lines with the `\` continuation character.
- Also can split long code lines in parentheses without using continuation characters (as in the script's first `print` statement)—this is preferred according to the *Style Guide for Python Code*.

Reading Integer Values from the User

- We use built-in functions `input` and `int` to prompt for and read two integer values from the user.

`if` Statements

- Each `if` statement consists of the keyword `if`, the condition to test, and a colon (`:`) followed by an indented body called a **suite**.
- Each suite must contain one or more statements.

Suite Indentation

- Python requires you to indent the statements in suites.
- The *Style Guide for Python Code* recommends four-space indents.

Confusing == and =

- Using the assignment symbol (=) instead of the equality operator (==) in an `if` statement's condition is a common syntax error.

Chaining Comparisons

- You can chain comparisons to check whether a value is in a range.

In [7]:

```
x = 3
```

In [8]:

```
1 <= x <= 5
```

Out[8]:

```
True
```

In [9]:

```
x = 10
```

In [10]:

```
1 <= x <= 5
```

Out[10]:

```
False
```

Precedence of the Operators We've Presented So Far

Operators				Grouping	Type
`()`				left to right	parentheses
`**`				right to left	exponentiation
`*`	`/`	`//`	`%`	left to right	multiplication, true division, floor division, remainder
`+`	`–`			left to right	addition, subtraction
`>`	`<=`	`<`	`>=`	left to right	less than, less than or equal, greater than, greater than or equal
`==`	`!=`			left to right	equal, not equal

2.8 Objects and Dynamic Typing

- `7` (an integer), `4.1` (a floating-point number) and `'dog'` are all objects.
- Every object has a type and a value.

In [1]:

```
type(7)
```

Out[1]:

```
int
```

In [2]:

```
type(4.1)
```

Out[2]:

```
float
```

In [3]:

```
type('dog')
```

Out[3]:

```
str
```

- An object's value is the data stored in the object.
- The snippets above show objects of built-in types **`int`** (for integers), **`float`** (for floating-point numbers) and **`str`** (for strings).

Variables Refer to Objects

- Assigning an object to a variable **binds** (associates) that variable's name to the object.
- Can then use the variable to access the object's value.

In [4]:

```
x = 7
```

In [5]:

```
x + 10
```

Out[5]:

```
17
```

In [6]:

```
x
```

Out[6]:

```
7
```

- Variable `x` **refers** to the integer object containing `7`.

In [7]:

```
x = x + 10
```

In [8]:

```
x
```

Out[8]:

```
17
```

Dynamic Typing (1 of 2)

- Python uses **dynamic typing**—it determines the type of the object a variable refers to while executing your code.
- Can show this by rebinding the variable `x` to different objects and checking their types.

Dynamic Typing (2 of 2)

In [9]:

```
type(x)
```

Out[9]:

```
int
```

In [10]:

```
x = 4.1
```

In [11]:

```
type(x)
```

Out[11]:

```
float
```

In [12]:

```
x = 'dog'
```

```
In [13]:
type(x)
Out[13]:
str
```

Garbage Collection

- Python creates objects in memory and removes them from memory as necessary.
- When an object is no longer bound to a variable, Python can automatically removes the object from memory.
- This process—called **garbage collection**—helps ensure that memory is available for new objects.

2.9 Intro to Data Science: Basic Descriptive Statistics

- In data science, you'll often use statistics to describe and summarize your data.
- Here, we introduce several **descriptive statistics**:
    - **minimum**—the smallest value in a collection of values.
    - **maximum**—the largest value in a collection of values.
    - **range**—the range of values from the minimum to the maximum.
    - **count**—the number of values in a collection.
    - **sum**—the total of the values in a collection.

Determining the Minimum of Three Values

- Determine the minimum of three values manually in a script that prompts for and inputs three values, uses `if` statements to determine the minimum value, then displays it.

```python
# fig02_02.py
"""Find the minimum of three values."""

number1 = int(input('Enter first integer: '))
number2 = int(input('Enter second integer: '))
number3 = int(input('Enter third integer: '))

minimum = number1

if number2 < minimum:
    minimum = number2

if number3 < minimum:
    minimum = number3

print('Minimum value is', minimum)
```

- Enter `12` , `27` and `36`

In [1]:

```
run fig02_02.py
```

```
Minimum value is 12
```

- Enter `12` , `36` and `27`

In [2]:

```
run fig02_02.py
```

```
Minimum value is 12
```

- Enter `36`, `27` and `12`

In [3]:

```
run fig02_02.py
```

```
Minimum value is 12
```

- Logic of determining minimum:
    - First, assume that `number1` contains the smallest value.
    - The first `if` statement then tests `number2 < minimum` and if this condition is `True` assigns `number2` to `minimum`.
    - The second `if` statement then tests `number3 < minimum`, and if this condition is `True` assigns `number3` to `minimum`.
- Now, `minimum` contains the smallest value, so we display it.

Determining the Minimum and Maximum with Built-In Functions `min` and `max` (1 of 2)

- Python has many built-in functions for performing common tasks.
- **`min`** and **`max`** calculate the minimum and maximum, respectively, of a collection of values.

Determining the Minimum and Maximum with Built-In Functions `min` and `max` (2 of 2)

In [4]:

```
min(36, 27, 12)
```

Out[4]:

```
12
```

In [5]:

```
max(36, 27, 12)
```

Out[5]:

```
36
```

Determining the Range of a Collection of Values (1 of 2)

- The *range* of values is simply the minimum through the maximum value.
- Much data science is devoted to getting to know your data.

Determining the Range of a Collection of Values (2 of 2)

- Have to understand how to interpret statistics.
  - If you have 100 numbers with a range of 12 through 36, those numbers could be distributed evenly over that range.
  - At the opposite extreme, you could have clumping with 99 values of 12 and one 36, or one 12 and 99 values of 36.

Functional-Style Programming: Reduction

- We introduce various *functional-style programming* capabilities.
- These enable you to write code that can be more concise, clearer and easier to **debug**.
- Functions `min` and `max` are examples of a functional-style programming concept called **reduction**—each reduces a collection of values to a *single* value.
- Other reductions you'll see: sum, average, variance and standard deviation.