CS1010X: Programming Methodology I
Chapter 1: Introduction to Python

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 1: Introduction to Python

Learning Objectives

By the end of this chapter, you should be able to:

- **Identify the Elements of Programming:** Recognize primitives, means of combination, and means of abstraction.
- **Manipulate Primitive Data:** Work with integers, floats, strings, and booleans, and perform type conversions.
- **Apply Operators Correctly:** Use arithmetic, comparison, and logical operators, respecting precedence and associativity rules.
- **Understand String Operations:** Understand immutability, character encoding (ASCII), and perform string slicing.
- **Understand Control Flow:** Implement logic using conditional statements (`if`, `elif`, `else`).

1.1 What is Python?

Python is a **high-level, interpreted, general-purpose** programming language. It is also a **dynamically** and **strongly** typed language.

- **High-level:** It abstracts away low-level details like memory management, making it easier to read and write.
- **Interpreted:** Code is executed line-by-line by the Python interpreter, rather than being compiled into machine code all at once.
- **Dynamically Typed:** Type checking is performed at runtime. Variables do not have fixed types; they are just names binding to objects. You can reassign a variable from an integer to a string.
- **Strongly Typed:** The interpreter enforces type constraints rigorously. It will not implicitly convert types in ambiguous situations. For example, `"5" + 5` raises a `TypeError`, whereas a weakly typed language (like JavaScript) might output `"55"`.

Interpreted Language Implications

Note that since Python is an interpreted language, any code that will lead to an error that is not run will be ignored. For example,

```
def error():
  return 1 // 0 # This should lead to a ZeroDivisionError!

print(1 + 1) # But we never run it, so no errors are thrown!
```

The output above is simply 2.

1.2 The Elements of Programming

Every powerful language has three mechanisms for accomplishing this:

1. Primitive Expressions
The simplest entities the language is concerned with (e.g. numbers, strings).

2. Means of Combination
The ways by which compound elements are built from simpler ones (e.g. operators like $+$, $*$, `and`).

3. Means of Abstraction
The ways by which compound elements can be named and manipulated as units (e.g. variables, functions).

1.3 Primitives and Types

In Python, everything is an object, and every object has a **type**. You can check the type of an object using the `type()` function.

1.3.1 Common Primitive Types

Table 1.1: Primitives

Type	Name	Example
Integer	`int`	`42, -5, 0`
Floating Point	`float`	`3.14, 2.0, 1e5`
String	`str`	`"Hello", 'Python'`
Boolean	`bool`	`True, False`
NoneType	`NoneType`	`None`

1.3.2 Type Conversion

You can explicitly convert between types (casting) using type functions.

```
x = "123"        # This is a str
y = int(x)       # Converts "123" to integer 123
z = float(x)     # Converts "123" to float 123.0
s = str(45.6)    # Converts 45.6 to string "45.6"
```

Type Errors

Not all conversions are valid. Trying to convert a non-numeric string to a number (e.g. `int("hello")`) will raise a `ValueError`.

1.4 Means of Combination: Operators

1.4.1 Arithmetic Operators

Python supports standard arithmetic. Note the specific behavior of division.

```
print(10 + 3)   # Addition: 13
print(10 - 3)   # Subtraction: 7
print(10 * 3)   # Multiplication: 30
print(10 / 3)   # True Division: 3.3333... (Always returns float)
print(10 // 3)  # Floor Division: 3
print(10 % 3)   # Modulo (Remainder): 1
print(10 ** 3)  # Power: 1000
```

Negative Floor Division

In Python, the `//` operators always takes the **floor** of the division. This means

```
print(-1 // 3) # -1 / 3 = -0.33333, so it rounds down to -1.
```

In other words, `a//b` returns $\lfloor \frac{a}{b} \rfloor$.

1.4.2 Comparison Operators

Comparisons return a Boolean value (`True` or `False`).

```
x = 10
y = 20
print(x == y)   # Equal to (False)
print(x != y)   # Not equal to (True)
print(x < y)    # Less than (True)
print(x >= y)   # Greater than or equal to (False)
```

String Comparison

Strings are compared character by character based on their **ASCII/Unicode** values. This is known as a *lexicographical order*.

- **Order:** Uppercase letters come *before* lowercase letters (e.g. 'Z' < 'a').
- **Process:** Python compares the first character; if they differ, that determines the result. If they are the same, it moves to the next.

```
print("apple" < "banana")  # True ('a' < 'b')
print("apple" == "Apple")  # False (Case sensitive)
print("Zebra" < "apple")   # True ('Z' is 90, 'a' is 97)
print("abcd" < "abcde")    # True (Prefix is smaller than full string)
```

Cross-Type Comparisons

- **Equality (==):** Generally safe. `5 == "5"` is `False` (they are different types).
- **Ordering (<, >):** In Python 3, comparing different types (e.g. `5 < "5"`) raises a `TypeError`. You cannot meaningfully order a number against a string.

1.4.3 Logical Operators

Python uses English keywords: `and`, `or`, `not`.

Truth Tables

Table 1.2: Truth Table

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

Short-Circuit Evaluation & "Truthiness"

In Python, logical operators do not just return Booleans; they return **one of the operands**.

- `x or y`: If x is "truthy", return x. Else, return y.
- `x and y`: If x is "falsy", return x. Else, return y.

What is False?

In Python, the following are considered "Falsy": `False`, `None`, `0`, `0.0`, `""` (empty string), `[]` (empty list). Almost everything else is "Truthy".

In Python, *short-circuiting* happens. That is, once the condition for the logical operator is fulfilled, it will immediately return the value that fulfilled it.

```python
# Example 1: OR (Returns first truthy value)
print(True or 20)      # Output: True
print(20 or True)      # Output: 20 (20 is truthy, short-circuit happens)
print(0 or 50)         # Output: 50 (0 is falsy, checks next)

# Example 2: AND (Returns first falsy value, or the last value)
print(12 and False)    # Output: False
print(0 and False)     # Output: 0
print(10 and 20)       # Output: 20 (10 is truthy, proceeds to 20)

# Example 3: Mixed
# 'not False' is True. 'True and 12' returns 12.
print(not False and 12) # Output: 12
```

1.4.4 Order of Operations

When an expression contains multiple operators, Python follows a strict order of precedence (similar to standard mathematics).

Table 1.3: Overview of order of operations

Precedence	Operator	Description	Associativity
1	`()`	Parentheses	Left-to-right
2	`**`	Exponentiation	Right-to-left
3	`+x, -x`	Positive, negative	Right-to-left
4	`*, /, \%, //`	Multiplication, division, modulus, floor division	Left-to-right
5	`+, -`	Addition, subtraction	Left-to-right
6	`<<, >>`	Bitwise shift left, bitwise shift right	Left-to-right
7	`&`	Bitwise AND	Left-to-right
8	`^`	Bitwise XOR	Left-to-right
9	`|`	Bitwise OR	Left-to-right
10	`<, <=, >, >=`	Comparison operators	Left-to-right
11	`==, !=`	Equality operators	Left-to-right
12	`not`	Logical NOT	Right-to-left
13	`and`	Logical AND	Left-to-right
14	`or`	Logical OR	Left-to-right
15	`if-else`	Ternary conditional	Right-to-left
16	`=, +=, -=, *=, /=, //=, \%=, **=, <<=, >>=, &=, ^=, |=`	Assignment and compound assignment	Right-to-left

1.5 Means of Abstraction: Variables

Variables allow us to bind values to names.

```
radius = 10           # Assignment
area = 3.14 * radius * radius
```

1.5.1 Naming Rules

- Must start with a letter (a-z, A-Z) or underscore (_).
- Can contain numbers, but cannot start with one.
- Case-sensitive (`age` is different from `Age`).
- Cannot be a Python keyword (e.g. `if`, `def`, `class`).

Convention

Python uses **snake_case** for variable names (e.g. `my_variable`, `student_score`). Avoid `camelCase`.

1.6 Strings

Strings are **immutable** sequences of characters enclosed in single (`' '`) or double (`" "`) quotes. Multiline strings are a thing in Python, if you enclose the text within three single or double quotes.

- **Immutable:** The object cannot be modified (*mutated*) after creation.

For example,

```
string = "abc"
string[0] = "b"
print(string) # An error is thrown, see below.

'''
Traceback (most recent call last):
  File "/tmp/exec_project_2927618818/code/main.py", line 2, in <module>
    string[0] = "b"
    ~~~~~~^^^
TypeError: 'str' object does not support item assignment
'''
```

1.6.1 Characters and Encoding

In Python, a single character is simply a string of length 1. Computers store characters as numbers using encoding standards like **ASCII** or **Unicode**.

- `ord(char)`: Returns the integer ASCII equivalent for a character.
- `chr(int)`: Returns the character for a specific ASCII code.

```
print(ord('A'))  # 65
print(ord('a'))  # 97
print(chr(66))   # 'B'
# Comparison relies on these values:
print('A' < 'a') # True (because 65 < 97)
```

Table 1.4: ASCII Printable Characters (32 – 126)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	(Spc)	56	38	8	80	50	P	104	68	h
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	$	60	3C	<	84	54	T	108	6C	l
37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	’	63	3F	?	87	57	W	111	6F	o
40	28	(	64	40	@	88	58	X	112	70	p
41	29	)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[	115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D	]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	’	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	|
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g			

Some notes about the ASCII table:

- The decimal value difference between an uppercase letter and its lowercase equivalent is 32 (e.g. ’A’ is 65 and ’a’ is 97). This is consistent across the alphabet.

1.6.2 String Slicing

In general, indexing on strings in python will be done using `s[x]`, where x is the index to sample, 0-indexed. This is done in $O(1)$ time.

However, slicing on strings involves the creation of new strings since they are immutable. Thus, they have a worst case time complexity of $O(n)$ time.

The concept of *Big-O Notation* ($O(f(x))$) will be introduced in subsequent lectures.

Slicing strings in Python is done by doing:

```
substring = s[start : end : step]
```

where the parameters are:

- `s`: The original string.
- `start` (optional): Starting index (inclusive). Defaults to 0 if omitted.
- `end` (optional): Stopping index (exclusive). Defaults to the end of the string if omitted.
- `step` (optional): Interval between indices. A positive value slices from left to right, while a negative value slices from right to left. If omitted, it defaults to 1 (no skipping of characters).

Negative indexing is useful for accessing elements from the end of the string. The last element has an index of -1, the second last element -2, and so on.

For example,

```
 Index:   0   1   2   3   4   5
        +---+---+---+---+---+---+
 Char:  | P | y | t | h | o | n |
        +---+---+---+---+---+---+
 Neg:    -6  -5  -4  -3  -2  -1
```

Basic usage of slices

In a slice, the start and stop positions for the subsequence are denoted as `[start:stop]`. The extracted range is `start` $\leq x <$ `stop`, including the item at `start` but excluding the item at `stop`.

```
s = "0123456"

print(s[2:5]) # '234' (Indices 2, 3, 4)
```

This example demonstrates slices for strings, but the same concept applies to lists, tuples, and other sequence objects. The case of negative values is described later.

Omitting `start` extracts the subsequence from the beginning, while omitting `stop` extracts it to the end. If both are omitted, all items are extracted.

```
s = "0123456"

print(s[:3])    # '012'
print(s[3:])    # '3456'
print(s[:])     # '0123456' (Full copy)
```

Out of range

Unlike simple indexing (which raises an `IndexError`), slicing handles out-of-range indices gracefully.

```
s = "0123456"
print(s[2:100]) # Output: '23456'; Stop is past end, stops at actual end
print(s[100:])  # Output: ''; Start is past end, returns empty string
```

No error occurs if you specify start and stop that select no item. An empty list is returned.

```
print(l[5:2])   # []
print(l[2:2])   # []
print(l[10:20]) # []
```

Steps

You can specify step in addition to start and stop as `[start:stop:step]`. For example, if step is set to 2, you can select items at odd-numbered or even-numbered positions.

```
s = "0123456"
print(s[::2])   # '0246' (Even positions)
print(s[1::2])  # '135'  (Odd positions)
```

As in the previous examples, if step is omitted, it is set to 1.

Extract from the end with a negative value

Negative values for start and stop represent positions from the end.

```
s = "Python"
# Index:   0   1   2   3   4   5
#        +---+---+---+---+---+---+
# Char:  | P | y | t | h | o | n |
#        +---+---+---+---+---+---+
# Neg:    -6  -5  -4  -3  -2  -1

print(s[-3:-1])    # 'ho'; Start at -3 (h), Stop at -1 (n), Step 1
print(s[:-2:-1])   # 'n'; Start at end, Stop BEFORE -2 (o), Step -1
print(s[-1:-4:-1]) # 'noh'; Start at -1 (n), Stop BEFORE -4 (t), Step -1
print(s[-1::])     # 'n'; Start at -1 (n), go to end, Step 1
print(s[-1::-1])   # 'nohtyP'; Start at -1 (n), go to beginning, Step -1
print(s[::-1])     # 'nohtyP'; Equivalent to above (Reversing a string)

# Direction Conflict (IMPORTANT)
# Start (-1) is to the right of Stop (1).
# Step is positive (defaults to 1), so it tries to go right.
# Result is empty because it can't reach index 1 by going right from -1.
print(s[-1:1])     # ''

# To fix the above, you must reverse the step:
print(s[-1:1:-1])  # 'noht'
```

Create slice objects with `slice()`

You can create a slice object using the built-in slice() function. To repeatedly select the items at the same position, you can create the slice object once and reuse it. Importantly, `slice(start, stop, step)` is equivalent to `start:stop:step`.

```
s = "0123456"
# Equivalent to s[2:5:2]
my_slice = slice(2, 5, 2)
print(s[my_slice]) # '24'
```

If two arguments are specified, step is set to None. This is equivalent to `start:stop`.

```
# Equivalent to s[start:stop]
sl_stop = slice(2,5)
print(s[sl_stop])  # '234'
```

If only one argument is specified, start and step are set to None. This is equivalent to `:stop`.

```
# Equivalent to s[:stop]
sl_stop = slice(5)
print(s[sl_stop])  # '01234'
```

If all arguments are omitted, a TypeError is raised. If you want to create a slice object equivalent to `:`, explicitly specify `None`.

```
# sl = slice()
# TypeError: slice expected at least 1 arguments, got 0

# Equivalent to s[:] (Full copy)
sl_all = slice(None)
print(s[sl_all])   # '0123456'
```

1.7 Control Flow: Conditionals

Conditional statements allow the program to execute different branches of code based on logic. There are three forms.

1.7.1 Form 1: Single If

Executed only if the condition is true.

```
if score >= 50:
    print("Pass")
```

1.7.2 Form 2: If-Else

Executed if true, otherwise execute the else block.

```
if score >= 50:
    print("Pass")
else:
    print("Fail")
```

1.7.3 Form 3: Chained Conditional (If-Elif-Else)

Checks conditions in order. Only the first true block runs.

```
if score >= 80:
    print("A")
elif score >= 70:
    print("B")
elif score >= 60:
    print("C")
else:
    print("F")
```