Chapter 1: Python Basics

Introduction

- Programs consist of statements to be run one after the other. A **statement** describes some action to be carried out.

Input/output

- **Basic output**
  - The **print()** function displays output to the user. **Output** is the information or result produced by a program.
  - The *sep* and *end* options can be used to customize the output
- Multiple values, separated by commas, can be printed in the same statement. By default, each value is separated by a space character in the output.
  - The *sep* option can be used to change this behavior.
- By default, the ***print()*** function adds a newline character at the end of the output
  - The *end* option can be used to continue printing on the same line.

Input/output

Code	Output
`print("Today is Monday.")` `print("I like string beans.")`	`Today is Monday.` `I like string beans.`
`print("Today", "is", "Monday")` `print("Today", "is", "Monday", sep="...")`	`Today is Monday` `Today...is...Monday`
`print("Today is Monday, ", end="")` `print("I like string beans.")`	`Today is Monday, I like string beans.`
`print("Today", "is", "Monday", sep="? ", end="!!")` `print("I like string beans.")`	`Today? is? Monday!!I like string beans.`

Input/output

1. Which line of code prints `Hello world!` as one line of output?
   a. `print(Hello world!)`
   b. `print("Hello", "world", "!")`
   c. `print("Hello world!")`

2. Which lines of code prints `Hello world!` as one line of output?
   a. `print("Hello")`
      `print(" world!")`
   b. `print("Hello")`
      `print(" world!", end="")`
   c. `print("Hello", end="")`
      `print(" world!")`

Input/output

Basic input

- Computer programs often receive input from the user.
- **Input** is what a user enters into a program.
- An input statement, ***variable = input("prompt")***, has three parts:
  - A **variable** refers to a value stored in memory.
  - The **input()** function reads one line of input from the user. A function is a named, reusable block of code that performs a task when called. The input is stored in the computer's memory and can be accessed later using the variable.
  - A **prompt** is a short message that indicates the program is waiting for input.

Input/output

4. Which line of code correctly obtains and stores user input?
   a. `input()`
   b. `today_is = input`
   c. `today_is = input()`

6. What is the output if the user enters "six" as the input?

```
print("Please enter a number: ")
number = input()
print("Value =", number)
```

   a. `Value = six`
   b. `Value = 6`
   c. `Value = number`

Data Types

- When we create a data item we can either assign it to a variable, or insert it into a collection.
- Variables allow programs to refer to values using names rather than memory locations.
- When we assign in Python, what really happens is that we bind an object reference to refer to the object in memory that holds the data.
- The names we give to our object references are called ***identifiers*** or just plain ***names***.
- A valid Python identifier is a nonempty sequence of characters of any length that consists of a "***start character***" and zero or more "***continuation characters***".

Data Types

- The start character can be anything that Unicode considers to be a letter, including the ASCII letters (“a”, “b”, ..., “z”, “A”, “B”, ..., “Z”), the underscore (“_”), as well as the letters from most non-English languages.
- Each continuation character can be any character that is permitted as a start character, or pretty well any nonwhite space character, including any character that Unicode considers to be a digit, such as (“0”, “1”, ..., “9”), or the Catalan character “·”.
- Example: ***TAXRATE, Taxrate, TaxRate, taxRate,*** and ***taxrate*** are five different identifiers.

```
π = 3.14            # Greek letter
नमस्ते = "Hello"    # Hindi
变量 = 42            # Chinese
_underscore = 10    # Underscore also allowed
```

Data Types

- The second rule is that no identifier can have the same name as one of Python's keywords

and	continue	except	global	lambda	pass	while
as	def	False	if	None	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	True	
class	else	from	is	or	try	

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Data Types

- Names that begin and end with two underscores (such as __lt__) should not be used. Python defines various special methods and variables that use such names

Integral Types

- Python provides two built-in integral types, **int** and **bool**.
- Both integers and Booleans are ***immutable***
- When used in Boolean expressions, 0 and False are False, and any other integer and True are True.
- When used in numerical expressions True evaluates to 1 and False to 0.

```
x = 5
print(id(x))  # e.g. 140711053364144
x = x + 1
print(x)      # 6
print(id(x))  # e.g. 140711053364176 → different from previous
```

Integral Types

- The size of an integer is limited only by the machine's memory
- **C and java** has limitation (32 bits)
- Integer literals are written using base 10 (decimal) by default

```
i = 5
print("Before:", i, "| id(i):", id(i))

i += True
print("After: ", i, "| id(i):", id(i))
```

```
>>> 14600926                          # decimal
14600926
>>> 0b110111101100101011011110        # binary
14600926
>>> 0o67545336                        # octal
14600926
>>> 0xDECADE                          # hexadecimal
14600926
```

Syntax	Description
`x + y`	Adds number `x` and number `y`
`x - y`	Subtracts `y` from `x`
`x * y`	Multiplies `x` by `y`
`x / y`	Divides `x` by `y`; always produces a `float` (or a `complex` if `x` or `y` is `complex`)
`x // y`	Divides `x` by `y`; truncates any fractional part so always produces an `int` result; see also the `round()` function
`x % y`	Produces the modulus (remainder) of dividing `x` by `y`
`x ** y`	Raises `x` to the power of `y`; see also the `pow()` functions
`-x`	Negates `x`; changes `x`'s sign if nonzero, does nothing if zero
`+x`	Does nothing; is sometimes used to clarify code
`abs(x)`	Returns the absolute value of `x`
`divmod(x, y)`	Returns the quotient and remainder of dividing `x` by `y` as a tuple of two `int`s
`pow(x, y)`	Raises `x` to the power of `y`; the same as the `**` operator
`pow(x, y, z)`	A faster alternative to `(x ** y) % z`
`round(x, n)`	Returns `x` rounded to *n* integral digits if *n* is a negative `int` or returns `x` rounded to *n* decimal places if *n* is a positive `int`; the returned value has the same type as `x`; see the text

```
print(round(3.14159, 2))
print(round(3.14159, 4))
print(round(3.14159))
print(round(1500, -2))
print(round(1689, -2))
```

Syntax	Description
`bin(i)`	Returns the binary representation of `int i` as a string, e.g., `bin(1980) == '0b11110111100'`
`hex(i)`	Returns the hexadecimal representation of `i` as a string, e.g., `hex(1980) == '0x7bc'`
`int(x)`	Converts object `x` to an integer; raises `ValueError` on failure—or `TypeError` if `x`'s data type does not support integer conversion. If `x` is a floating-point number it is truncated.
`int(s, base)`	Converts `str s` to an integer; raises `ValueError` on failure. If the optional *`base`* argument is given it should be an integer between 2 and 36 inclusive.
`oct(i)`	Returns the octal representation of `i` as a string, e.g., `oct(1980) == '0o3674'`

Integral Types

- All the binary numeric operators ***(+, -, /, //, %, and **)*** have augmented assignment versions ***(+=, -=, /=, //=, %=, and **=)*** where **x *op*= y** is logically equivalent to **x = x *op* y** in the normal case when reading x's value has no side effects.

- All the binary bitwise operators ***(|, ^, &, <<, and >>)*** have augmented assignment versions ***(|=, ^=, &=, <<=, and >>=)*** where **i *op*= j** is logically equivalent to **i = i *op* j** in the normal case when reading i's value has no side effects.

Integral Types

Syntax	Description
`i | j`	Bitwise OR of `int i` and `int j`; negative numbers are assumed to be represented using 2's complement
`i ^ j`	Bitwise XOR (exclusive or) of `i` and `j`
`i & j`	Bitwise AND of `i` and `j`
`i << j`	Shifts `i` left by `j` bits; like `i * (2 ** j)` without overflow checking
`i >> j`	Shifts `i` right by `j` bits; like `i // (2 ** j)` without overflow checking
`~i`	Inverts i's bits

Integral Types

- Boolean:
  - There are two built-in Boolean objects: True and False

```
# Case 1: No arguments
print("bool() →", bool())  # Returns False

# Case 2: Passing a bool
a = True
print("bool(True) →", bool(a))  # Returns True (copy of the bool)

# Case 3: Passing other types
print("bool(0) →", bool(0))       # False (0 is considered False)
print("bool(1) →", bool(1))       # True
print("bool('') →", bool(''))     # False (empty string is False)
print("bool('hello') →", bool('hello'))  # True
print("bool([]) →", bool([]))     # False (empty list)
print("bool([1, 2, 3]) →", bool([1, 2, 3]))  # True (non-empty list)
```

Floating-Point Types

- Python provides three kinds of floating-point values: the built-in **float** and **complex types**,and the **decimal.Decimal** type from the standard library
- All three are immutable.
- Type float holds double-precision floating-point numbers whose range depends on the C
- Numbers of type float are written with a decimal point, or using exponential notation
  - for example, 0.0, 4., 5.7, -2.5, -2e9, 8.9e-4.

Floating-Point Types

- The float data type can be called as a function
  - with no arguments it returns 0.0
  - with a float argument it returns a copy of the argument
  - and with any other argument it attempts to convert the given object to a float.
- Floating-point numbers can be converted to integers using the
  - int()
  - round()
  - math.floor() or math.ceil()
- The ***float.is_integer()*** method returns **True** if a floating-point number's fractional part is 0

Floating-Point Types

```
print("float() →", float())                    # No arguments → 0.0
print("float(3.14) →", float(3.14))            # With float → returns same
print("float('2.718') →", float('2.718'))      # String → converted to float

x = 5.75
print("int(x) →", int(x))                      # Truncates decimal
print("round(x) →", round(x))                  # Rounds to nearest int
print("math.floor(x) →", math.floor(x))        # Down
print("math.ceil(x) →", math.ceil(x))          #Up
```

Floating-Point Types

```
print("float(3.0).is_integer() →", float(3.0).is_integer())  # True
print("float(3.14).is_integer() →", float(3.14).is_integer())# False
```

Floating-Point Types

- The complex data type is an immutable type that holds a pair of floats, one representing the real part and the other the imaginary part of a complex number.
- Literal complex numbers are written with the real and imaginary parts joined by a **+** or **-** sign, and with the imaginary part followed by a **j**.
- Here are some examples: 3.5+2j, 0.5j, 4+0j, -1-3.7j.
- Notice that if the real part is 0, we can omit it entirely.

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

Floating-Point Types

- The decimal module provides immutable Decimal numbers that are as accurate as we specify.
- Calculations involving Decimals are slower than those involving floats

```
>>> import decimal
>>> a = decimal.Decimal(9876)

>>> b = decimal.Decimal("54321.01234567898765432l")
>>> a + b
Decimal('64197.01234567898765432l')
```

Strings

- Strings are represented by the ***immutable str*** data type which holds a sequence of Unicode characters.
- **string objects**—with *no arguments* it returns an empty string, with a *nonstring argument* it returns the string form of the argument, and with a *string argument* it returns a copy of the string.
- Strings are generally defined using **single** or **double** quotes
- In addition, we can use a ***triple quoted string***

```
text = """A triple quoted string like this can include 'quotes' and
"quotes" without formality. We can also escape newlines \
so this particular string is actually only two lines long."""
```

Strings

- If we want to use quotes inside a normal quoted string we can do so without formality if they are different from the delimiting quotes; otherwise, we must escape them:

```
a = "Single 'quotes' are fine; \"doubles\" must be escaped."
b = 'Single \'quotes\' must be escaped; "doubles" are fine.'
```

```
import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

Strings

- The solution is to use ***raw*** strings. These are quoted or triple quoted strings whose first quote is preceded by the letter *r*. Inside such strings all characters are taken to be literals, so no escaping is necessary.

```
phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:-\d+)?)$")
```

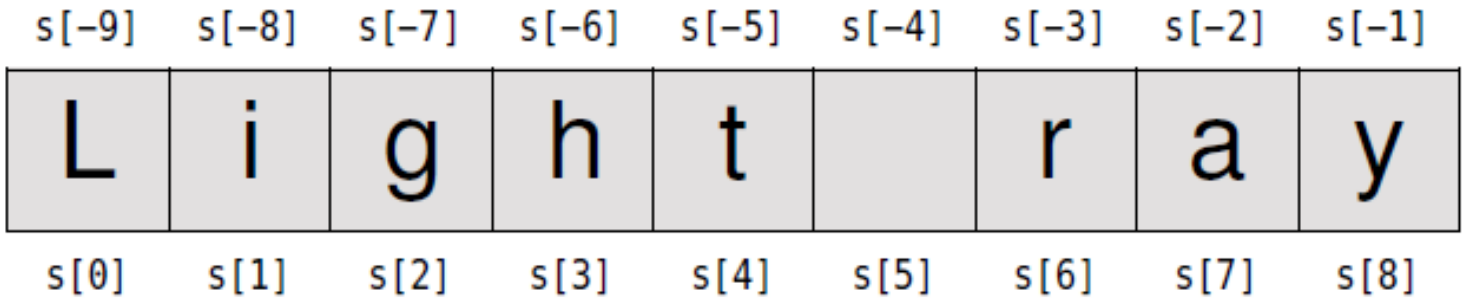
- If we want to write a long string literal spread over two or more lines but without using a triple quoted string there are a couple of approaches we can take:

```
t = "This is not the best way to join two long strings " + \
    "together since it relies on ugly newline escaping"

s = ("This is the nice way to join two long strings "
     "together; it relies on string literal concatenation.")
```

Slicing and Striding Strings

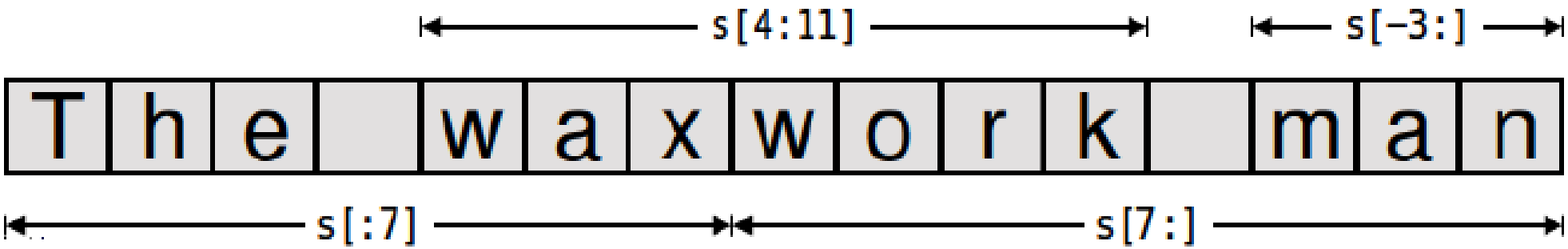
- Individual characters in a string, can be extracted using the item access operator ([])
- Index positions into a string begin at 0 and go up to the length of the string minus 1. But it is also possible to use negative index positions—these count from the last character back toward the first.



Slicing and Striding Strings

- The slice operator has three syntaxes:
  - ***seq*[*start*]**
  - ***seq*[*start*:*end*]**
  - ***seq*[*start*:*end*:*step*]**
- The *seq* can be any sequence, such as a list, string, or tuple. The *start*, *end*, and *step* values must all be integers (or variables holding integers)
- The second syntax extracts a slice from and including the *start*-th item, up to and *excluding* the *end*-th item.

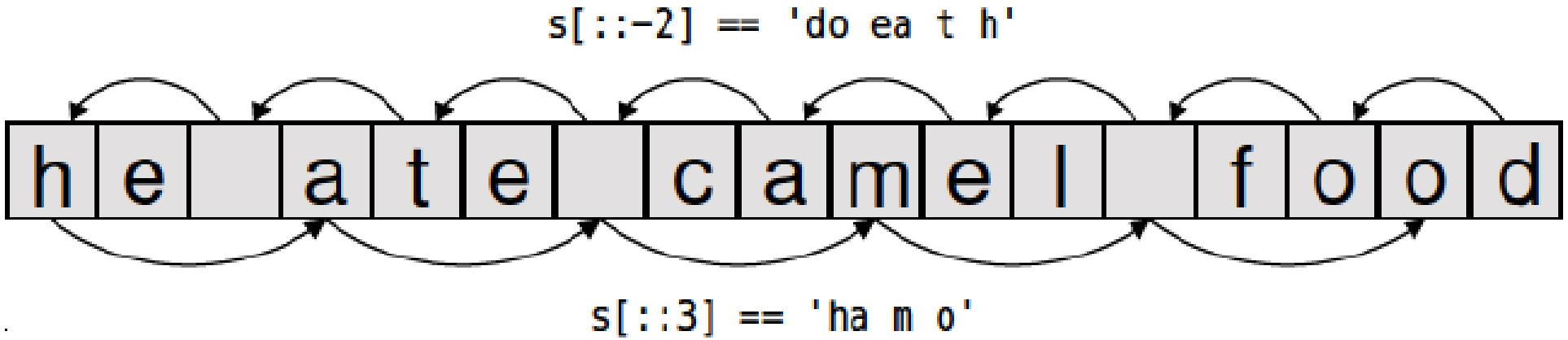
Slicing and Striding Strings




```
>>> s = s[:12] + "wo" + s[12:]
>>> s
```

```
    s[:12] + s[7:9] + s[12:].
```

Slicing and Striding Strings



String: Case Conversion

Method	Description
s.upper()	Converts all characters to uppercase
s.lower()	Converts all characters to lowercase
s.capitalize()	Capitalizes the first character
s.title()	Capitalizes the first character of each word
s.swapcase()	Swaps case (upper to lower and vice versa)

String: Searching and Finding

Method	Description
s.find(sub)	Returns the index of first occurrence of sub, or -1
s.rfind(sub)	Returns the index of last occurrence of sub, or -1
s.index(sub)	Like find(), but raises ValueError if not found
s.count(sub)	Counts how many times sub occurs

String: Check String Properties

Method	Description
s.isalpha()	True if all characters are letters
s.isdigit()	True if all characters are digits
s.isalnum()	True if all characters are alphanumeric
s.isspace()	True if all characters are whitespace
s.islower() / s.isupper()	True if all characters are lower/upper
s.startswith(sub)	True if string starts with sub
s.endswith(sub)	True if string ends with sub

String: Modifying and Formatting

Method	Description
s.replace(old, new)	Replaces all occurrences of old with new
s.strip()	Removes leading and trailing whitespace
s.lstrip() / s.rstrip()	Removes whitespace from left/right
s.split(sep)	Splits the string into a list (default: whitespace)
s.join(iterable)	Joins elements of iterable using s as separator
s.zfill(width)	Pads the string with zeros on the left
s.center(width)	Centers the string in a field of given width

Strings: Examples

```
s = "banana"
index = s.rfind("a")
print(index)
```

```
s = "banana"
print(s.rfind("z"))
```

```
s = "hello world"
print(s.index("o"))
```

```
s = "hello world"
print(s.index("z"))
```

```
s = "abc123"
print(s.isalnum())
```

```
s = "abc 123"
print(s.isalnum())
```

```
s = "abc@123"
print(s.isalnum())
```

```
s = "  hello world  "
print(s.strip())
# Output: "hello world"
```

Strings: Examples

```
s = "##Hello##"
print(s.strip("#"))
```

```
s = "  hello  world  "
print(s.strip())
# Output: "hello  world"
```

```
items = ["apple", "banana", "cherry"]
result = ", ".join(items)
print(result)
```

Strings: Examples

```
s = "hello world"
char = 'l'
first = s.find(char)
second = s.find(char, first + 1)
print("Second occurrence:", second)
```

```
s = "Python is fun"
words = s.split()
r_words = ' '.join(words[::-1])
print(r_words)
```

7. What is the output of the following code?

```
number = "12"
number_of_digits = len(number)
print("Number", number, "has", number_of_digits, "digits.")
```

a. Number 12 has 12 digits.
b. Number 12 has 2 digits.
c. Number 12 has number_of_digits digits.

10. A user enters `"red"` after the following line of code.

```
color = input("What is your favorite color?")
```

Which produces the output `"Your favorite color is red!"`?

a. `print("Your favorite color is " + color + !)`
b. `print("Your favorite color is " + "color" + "!")`
c. `print("Your favorite color is " + color + "!")`

11. Which of the following assigns `"one-sided"` to the variable `holiday`?

a. `holiday = "one" + "sided"`
b. `holiday = one-sided`
c. `holiday = "one-" + "sided"`

Collection Data types

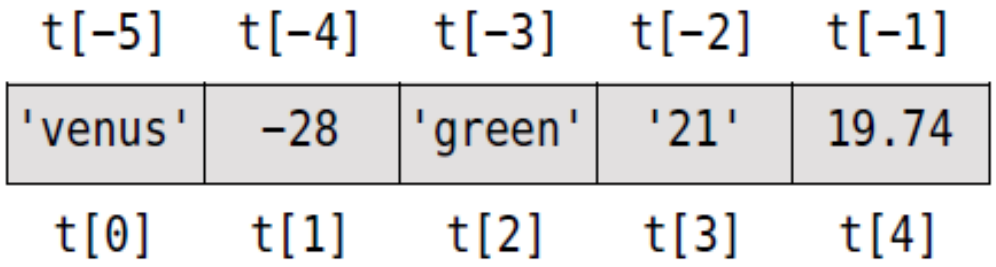
- A *sequence* type is one that supports the membership operator (in), the size function (len()), slices ([]), and is iterable.

Collection Data Types: Tuples

- A tuple is an **ordered sequence** of zero or more object references.
- Tuples support the same slicing and striding syntax as strings
- Like strings, tuples are ***immutable***, so we cannot replace or delete any of their items.
- The tuple data type can be called as a function, **tuple()**—
  - with no arguments it returns an empty tuple
  - with a tuple argument it returns a copy of the argument
  - and with any other argument it attempts to convert the given object to a tuple.

Collection Data Types: Tuples

- An empty tuple is created using empty parentheses, (), and a tuple of one or more items can be created by using commas.



- Tuples provide just two methods,
  - ***t*.count(*x*)**, which returns the number of times object *x* occurs in tuple *t*
  - ***t*.index(*x*)**, which returns the index position of the leftmost occurrence of object *x* in tuple *t*—or raises a Value Error exception if there is no *x* in the tuple.

Collection Data Types: Tuples

- Animals= “dog”, “cat”, “tiger”, “lion”
- Animals[1:]
- Animals[-3:]

- >> Animals[:2], “elephant”, Animals[2:]

- >>Animals[:2] + (“elephant”,)+ Animals[2:]

Collection Data Types: Tuples

```
>>> hair = "black", "brown", "blonde", "red"
>>> eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
>>> colors = (hair, eyes)
>>> colors[1][3:-1]
```

```
data = ("a", "b", "c", ("d", "e", "f"), "g", "h")
result = data[3][1:] + (data[1],) + data[-2:-5:-1]

print(result)
```

Collection Data Types: Tuples

```
nums = (10, 20, 30, 40, 50)
temp = nums[1:4]
try:
    temp[1] = 100
except TypeError as e:
    print("Error:", e)

reversed_tuple = nums[::-2]
print("Reversed:", reversed_tuple)
```

Collection Data types: Named Tuples

- A named tuple behaves just like a plain tuple, and has the same performance characteristics.
- What it adds is the ability to refer to items in the tuple by name as well as by index position
- The collections module provides the **namedtuple()** function. This function is used to create custom tuple data types. For example:

```
Sale = collections.namedtuple("Sale",
            "productid customerid date quantity price")
```

  - The first argument to **collections.namedtuple()** is the name of the custom tuple data type that we want to be created.
  - The second argument is a string of space separated names, one for each item that our custom tuples will take.

Collection Data types: Named Tuples

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
```

Sales[0][-1]

```
total = 0
for sale in sales:
    total += sale.quantity * sale.price
print("Total ${0:.2f}".format(total))  # prints: Total $42.46
```

Collection Data Types: Lists

- A list is an ordered sequence of zero or more object references.
- Lists support the same slicing and striding syntax as strings and tuples.
- Unlike strings and tuples, lists are mutable, so we can **replace** and **delete** any of their items.
- It is also possible to **insert**, **replace**, and **delete** slices of lists.

Collection Data Types: Lists

- The list data type can be called as a function, list()—
  - with no arguments it returns an empty list,
  - with a list argument it returns a shallow copy of the argument,
  - and with any other argument it attempts to convert the given object to a list. It does not accept more than one argument.

Given the assignment `L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]`, we get the list shown in Figure 3.2.

L[-6]	L[-5]	L[-4]	L[-3]	L[-2]	L[-1]
-17.5	'kilo'	49	'V'	['ram', 5, 'echo']	7
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

Syntax	Description
`L.append(x)`	Appends item `x` to the end of `list L`
`L.count(x)`	Returns the number of times item `x` occurs in `list L`
`L.extend(m)` `L += m`	Appends all of iterable `m`'s items to the end of `list L`; the operator `+=` does the same thing

`L.index(x,`
`  start,`
`  end)`
	Returns the index position of the leftmost occurrence of item `x` in `list L` (or in the *start*`:`*end* slice of `L`); otherwise, raises a `ValueError` exception

`L.insert(i, x)`	Inserts item `x` into `list L` at index position `int i`
`L.pop()`	Returns and removes the rightmost item of `list L`
`L.pop(i)`	Returns and removes the item at index position `int i` in `L`
`L.remove(x)`	Removes the leftmost occurrence of item `x` from `list L`, or raises a `ValueError` exception if `x` is not found
`L.reverse()`	Reverses `list L` in-place
`L.sort(...)`	Sorts `list L` in-place; this method accepts the same *key* and *reverse* optional arguments as the built-in `sorted()`

```
L = [1, 2, 3]
L.append([4, 5])
L.extend([6, 7])
L.pop()
print(L)
```

```
L = [20, 5, 15, 10]
L.sort()
L.reverse()
val = L.pop(1)
print("Value popped:", val)
print("Final List:", L)
```

```
L = ['a', 'b', 'c', 'd', 'b']
L.insert(2, 'x')
L.remove('b')
index_pos = L.index('b')
print(L)
print("Index of 'b':", index_pos)
```

Collection Data Types:

- A ***list comprehension*** is an expression and a loop with an optional condition enclosed in brackets where the loop is used to generate items for the list, and where the condition can filter out unwanted items

```
[item for item in iterable]
[expression for item in iterable]
[expression for item in iterable if condition]
```

```
words = ["apple", "banana", "cherry", "avocado"]
a_words = [word for word in words if word.startswith('a')]
print(a_words)
```

```python
numbers = [5, 12, 7, 20, 3]
for index, value in enumerate(numbers):
    if value % 2 == 0:
        print(f"Index {index} has even number {value}")
```

```python
marks = [45, 67, 89, 34, 76, 58]
passed = [m for m in marks if m >= 50]
print("Passed Marks:", passed)
```

Collection Data Types: Dictionaries

- A **dict** is an unordered collection of zero or more key–value pairs whose keys are object references that refer to hash table objects, and whose values are object references referring to objects of any type
- Dictionaries are **mutable**, so we can easily *add or remove* items, but since they are unordered they have no notion of index position and so cannot be sliced or strided.
- Dictionaries can also be created using braces—empty braces, {},
- Nonempty braces must contain one or more comma separated items, each of which consists of a key, a literal colon, and a value

Collection Data Types: Dictionaries

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

Collection Data Types: Dictionaries

- Because dictionaries have both keys and values, we might want to iterate over a dictionary by **(key, value) items**, by **values**, or by **keys**.

```
for item in d.items():
    print(item[0], item[1])
```

```
for key, value in d.items():
    print(key, value)
```

```
for value in d.values():
    print(value)
```

```
for key in d:
    print(key)
```

```
for key in d.keys():
    print(key)
```

Syntax	Description
`d.clear()`	Removes all items from `dict d`
`d.copy()`	Returns a shallow copy of `dict d`
`d.fromkeys(s, v)`	Returns a `dict` whose keys are the items in sequence `s` and whose values are `None` or *v* if *v* is given
`d.get(k)`	Returns key `k`'s associated value, or `None` if `k` isn't in `dict d`
`d.get(k, v)`	Returns key `k`'s associated value, or `v` if `k` isn't in `dict d`
`d.items()`	Returns a view* of all the (key, value) pairs in `dict d`
`d.keys()`	Returns a view* of all the keys in `dict d`
`d.pop(k)`	Returns key `k`'s associated value and removes the item whose key is `k`, or raises a `KeyError` exception if `k` isn't in `d`
`d.pop(k, v)`	Returns key `k`'s associated value and removes the item whose key is `k`, or returns `v` if `k` isn't in `dict d`
`d.popitem()`	Returns and removes an arbitrary (key, value) pair from `dict d`, or raises a `KeyError` exception if `d` is empty
`d.setdefault(k, v)`	The same as the `dict.get()` method, except that if the key is not in `dict d`, a new item is inserted with the key `k`, and with a value of `None` or of *v* if *v* is given
`d.update(a)`	Adds every (key, value) pair from `a` that isn't in `dict d` to `d`, and for every key that is in both `d` and `a`, replaces the corresponding value in `d` with the one in `a`—`a` can be a dictionary, an iterable of (key, value) pairs, or keyword arguments
`d.values()`	Returns a view* of all the values in `dict d`

```python
student = {'name': 'John', 'age': 21, 'marks': 85}
print(student['name'])
print(student.get('grade', 'Not Found'))
```

```python
colors = {'red': '#FF0000', 'green': '#00FF00', 'blue': '#0000FF'}
for color, hexcode in colors.items():
    print(f"{color} => {hexcode}")
```

```python
person = {'name': 'Alice', 'city': 'New York'}
person['age'] = 30
person['city'] = 'Boston'
del person['name']
print(person)
```

```python
nums = [1, 2, 3, 4]
square_dict = {n: n**2 for n in nums if n % 2 == 0}
print(square_dict)
```

```python
d = {'a': 1, 'b': 2, 'c': 3}
result = list(d.keys())[::-1]
print(result)
```

Control Structures and Functions

- Python provides conditional branching with ***if statements*** and looping with ***while*** and ***for ...in statements***.
- Python also has a ***conditional expression***—this is a kind of if statement that is Python's answer to the ternary operator (?:)

Control Structures and Functions

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

In some cases, we can reduce an `if ... else` statement down to a single *conditional expression*. The syntax for a conditional expression is:

```
expression1 if boolean_expression else expression2
```

Control Structures and Functions

```
offset = 20
if not sys.platform.startswith("win"):
    offset = 10
```

```
offset = 20 if sys.platform.startswith("win") else 10
```

```
width = 100 + 10 if margin else 0
```

```
width = 100 + (10 if margin else 0)
```

Control Structures and Functions (While loop)

- The else clause is optional. As long as the *boolean_expression* is True, the while block's suite is executed.
- If the *boolean_expression* is or becomes False, the loop terminates, and if the optional else clause is present, its suite is executed.
- else clause's suite is always executed if the loop terminates normally **(break, return and exception)**

```
while boolean_expression:
    while_suite
else:
    else_suite
```

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("While loop completed without break")
```

Control Structures and Functions (While loop)

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

Control Structures and Functions (For loop)

- The *expression* is normally either a single variable or a sequence of variables, usually in the form of a tuple. If a tuple or list is used for the *expression*, each item is unpacked into the *expression*'s items

```
for expression in iterable:
    for_suite
else:
    else_suite
```

```
for num in [1, 2, 3, 4, 5]:
    if num == 0:
        break
    print(num)
else:
    print("Loop completed without break")
```

Control Structures and Functions (For loop)

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    return index
```

Exception Handling

- **Catching and Raising Exceptions**
- Exceptions are caught using **try ...except blocks**, whose general syntax is:

```
try:
    try_suite
except exception_group1 as variable1:
    except_suite1
...
except exception_groupN as variableN:
    except_suiteN
else:
    else_suite
finally:
    finally_suite
```

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("No error occurred.")    #
finally:
    print("This always runs.")     #
```

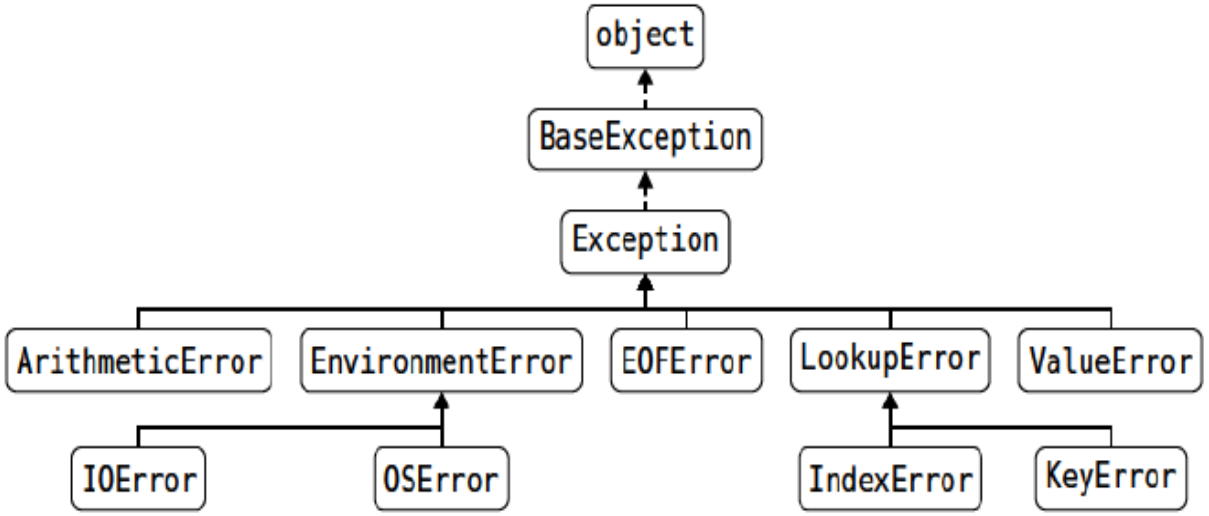
Exception Handling

- There must be at least one **except block**, but both the **else** and the **finally** blocks are optional.
- The else block’s suite is executed when the try block’s suite has finished normally—but it is not executed if an exception occurs.
- If there is a finally block, it is always executed at the end.

Exception Handling

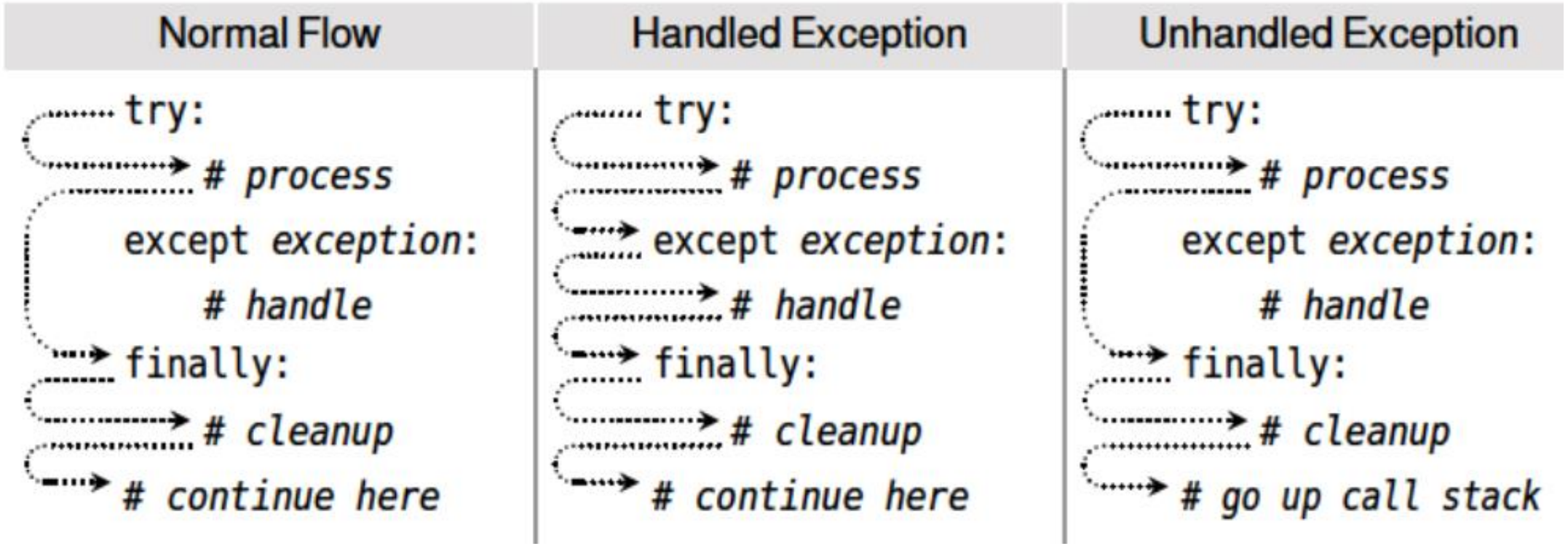
- Each except clause's exception group can be a single exception or a parenthesized tuple of exceptions.
- For each group, the ***as variable*** part is optional; if used, the variable contains the exception that occurred, and can be accessed in the exception block's suite.
- If an exception occurs in the try block's suite, each except clause is tried in turn.
- If the exception matches an exception group, the corresponding suite is executed.

Exception Handling




```
try:
    x = d[5]
except LookupError:    # WRONG ORDER
    print("Lookup error occurred")
except KeyError:
    print("Invalid key used")
```

Exception Handling



Exception Handling

```
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
    return lines
```

Example Programs

```
lst = [1, 2, 2, 3, 4, 4, 5]
unique = []
for i in lst:
    if i not in unique:
        unique.append(i)
print(unique)
```

```
s = "banana"
d = {}
for ch in s:
    d[ch] = d.get(ch, 0) + 1
print(d)
```

Example Programs

```
try:
    a = int("12a")
    b = 10 / a
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid conversion.")
except Exception as e:
    print("Other error:", e)
```

```
s = "madam"
is_pal = True
for i in range(len(s)//2):
    if s[i] != s[-(i+1)]:
        is_pal = False
        break
```

```
num = 1234
total = 0
while num > 0:
    total += num % 10
    num = num // 10
print(total)
```

Example Programs

```
data = ["12", "abc", "5", "0"]
for val in data:
    try:
        num = int(val)
        print(10 // num)
    except ZeroDivisionError:
        print("Divide by zero")
    except ValueError:
        print("Not a number")
```

Custom Functions

- Functions are a means by which we can package up and parameterize functionality.
- Four kinds of functions can be created in Python:
  - Global functions
  - Local functions
  - Lambda functions
  - Methods

Custom Functions

- ***Global functions*** are accessible to any code in the same module (i.e., the same .py file) in which the object is created.

- ***Local* functions** (also called nested functions) are functions that are defined inside other functions. These functions are visible only to the function where they are defined; they are especially useful for creating small helper functions that have no use elsewhere.
- ***Lambda* functions** are expressions, so they can be created at their point of use; however, they are much more limited than normal functions.

Custom Functions

- *Methods* are functions that are associated with a particular data type and can be used only in conjunction with the data type
- The general syntax for creating a (global or local) function is:

```
def functionName(parameters):
    suite
```

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Custom Functions

```
def shorten(text, length=25, indicator="..."):
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Here are a few example calls:

```
shorten("The Silkie")                          # returns: 'The Silkie'
shorten(length=7, text="The Silkie")           # returns: 'The ...'
shorten("The Silkie", indicator="&", length=7) # returns: 'The Si&'
shorten("The Silkie", 7, "&")                  # returns: 'The Si&'
```

Custom Functions

```
def append_if_even(x, lst=[]):
    if x % 2 == 0:
        lst.append(x)
    return lst
```

```
def append_if_even(x, lst=None):
    if lst is None:
        lst = []
    if x % 2 == 0:
        lst.append(x)
    return lst
```

- This idiom of using default none must be used for dict and list

Custom Functions

- We can also use the sequence unpacking operator in a function's parameter list. This is useful when we want to create functions that can take a variable number of positional arguments.

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

```
product(1, 2, 3, 4)     # args == (1, 2, 3, 4); returns: 24
product(5, 3, 8)        # args == (5, 3, 8); returns: 120
product(11)             # args == (11,); returns: 11
```

Custom Functions

- The function can be called with just positional arguments, for example,
  - **sum_of_powers(1, 3, 5)**,
- or with both positional and keyword arguments,
  - **sum_of_powers(1, 3, 5, power=2).**

```
def sum_of_powers(*args, power=1):
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

Custom Functions

```python
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))


def welcome_user(username):
    print(greet(username))

welcome_user("Bob")
```

```python
def outer_function(name):
    def add_greeting(text):
        return f"Hello, {text}!"
    return add_greeting(name)


print(outer_function("Charlie"))

print(add_greeting("Dave"))
```

Lambda Functions

- Lambda functions are functions created using the following syntax:

lambda *parameters*: *expression*

- The ***parameters*** are optional, and if supplied they are normally just comma separated variable names, that is, positional arguments
- Although the complete argument syntax supported by def statements can be used.
- The *expression* cannot contain branches or loops cannot have a return (or yield) statement
- The result of a lambda expression is an anonymous function. When a lambda function is called it returns the result of computing the *expression* as its result.

Lambda Functions

```
lambda x: x + 2
```

```
def add_two(x):
    return x + 2
```

```
lambda x:
    if x > 0:   #
        return x
    else:
        return -x
```

```
lambda x: x if x > 0 else -x
```

```
lambda lst: [x**2 for x in lst if x % 2 == 0]
```