Programming for Automation

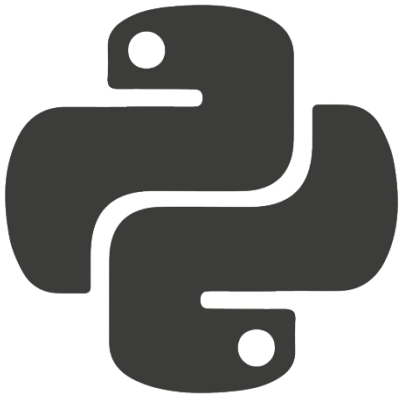
TEKsystems Global Services

WORKFORCE DEVELOPMENT

Python

Develop a passion for learning.

Python types



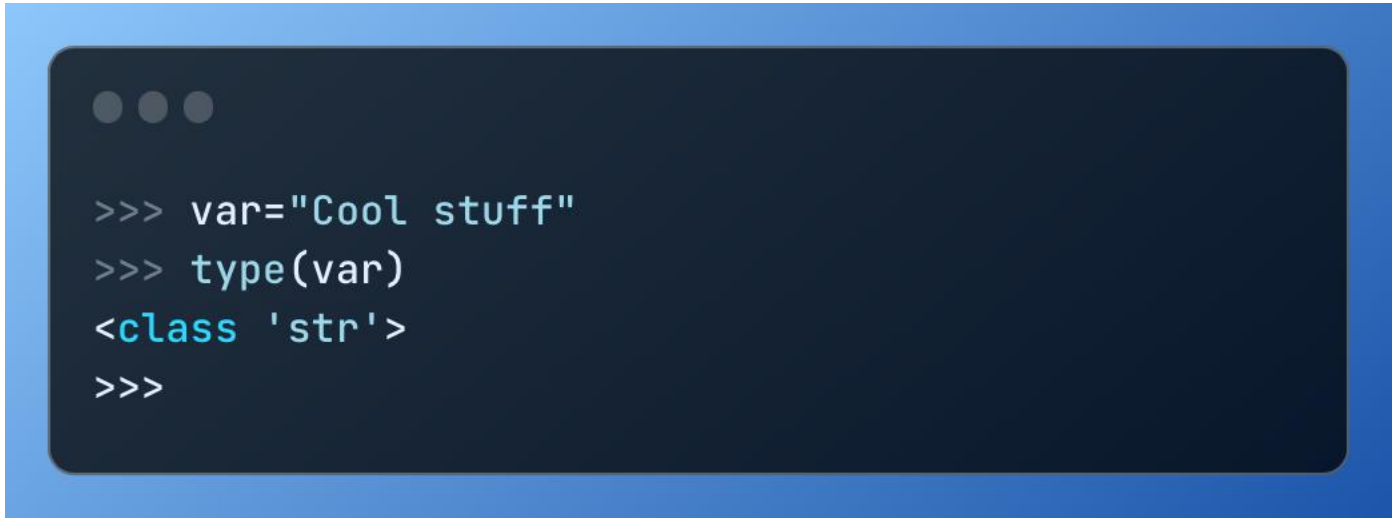
Variables are labels that are assigned to memory locations. Each memory location is a storage bin, which can contain data of a specific type.

- There are built-in and custom types
  - Integers, floats, strings, etc.
  - User-defined types (classes)
- Python is a dynamic language where variables are not assigned types declaratively.
- Nonetheless, Python is strongly typed.
- Objects can be either mutable or immutable. For example, lists are mutable. Sets and strings are immutable.

Python – declare a variable

It is simple to declare a variable in Python; assign a value!

- You do not specify a type, such as integer or float.
- The type is set based on the rvalue type varname=rvalue
- After the assignment, the variable is strongly typed



Python – declare a variable

5 MINUTES

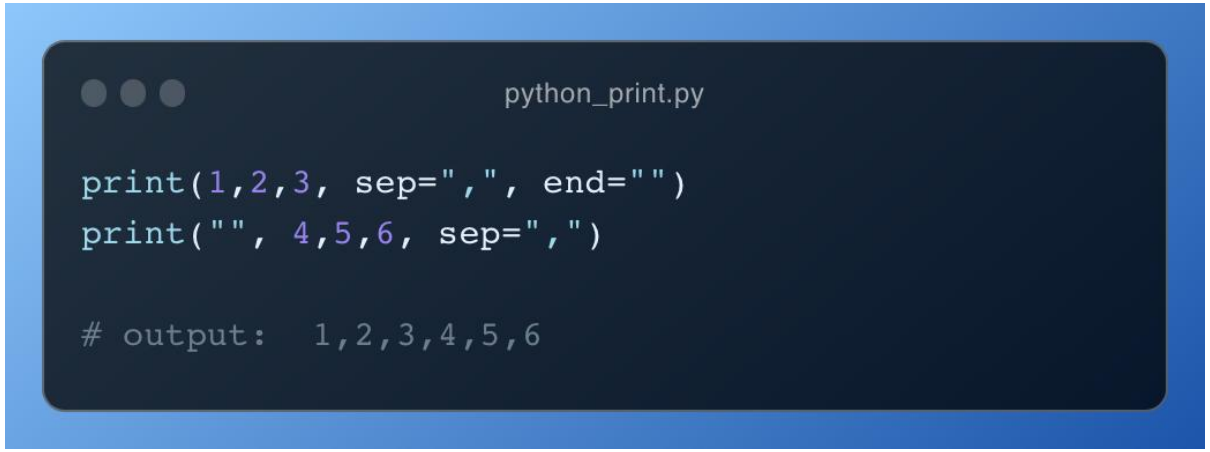
Open the Python Interpreter

1. Declare three variables as listed
    1. var1: 42 – Integer
    2. var2: 10.0 - Float
    3. var3: "Hello" – String
2. Use the type function to display the type of each variable.
3. Bonus: Try it with the “input” function and add error handling for non-numeric input.

Python - Print

You will often use the print command to validate your growing knowledge of Python.

- Variable length argument list
- Must be bounded with parentheses



Python - Print

The help page for print reveals that print can take any number of arguments (*args),. Print has other optional named parameters.

Having default values for the named parameters makes them optional.

```
>>> help(print)
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current
sys.stdout.
    flush
      whether to forcibly flush the stream.
```

Python - Print

You can also do basic formatting with the method: string.format .

Using parameter placeholders { n }, where n indicates parameter nth in the parameter list.

python_print_formatting.py

```
var1="ABC"
var2="DEF"
print("Forward {0}{1}".format(var1, var2))
print("Reverse {1}{0}".format(var1, var2))

#output:
#Forward ABCDEF
#Reverse DEFABC
```

Python 3.6+ fstring

In Python 3.6, **f-strings** were introduced to make string formatting simpler and more readable.

They allow you to embed variables and expressions directly inside strings.

f-strings are now the **preferred** way to format strings in modern Python.

Python 3.6+ fstring

f-strings are evaluated at runtime.
To use an f-string, simply place an **f** in front of the string.

f-strings support formatting options directly inside the braces and converts expressions to strings.

Python also allows strings to be defined as **bytes** by prefixing them with **b**.

```
>>> value = 7.489
...
... print(f"Formatted value: {value:.2f}")
...
... text = b"hello world"
... print(text)
...
Formatted value: 7.49
b'hello world'
```

Python - Input

The input function reads data from the console. The result is stored in a variable as a string.

The only parameter is a prompt. Use the prompt to provide instruction to the user.

Python - Input

In this code example, we use the input function to prompt for an age.

The int function will raise an exception if the string you enter cannot be converted to an int.

The only way out of the loop is to enter a string that can convert to an in.

```
>>> while True:
...     try:
...         age = int(input("Please enter your age: "))
...         break
...     except:
...         continue
...
Please enter your age: abv
Please enter your age: Steve
Please enter your age: steve1
Please enter your age: 44
>>> age
44
>>>
```

Python - Input

We call Split on the string. Split will return a list of substrings.

The first_name will be the first list item, and all others will be packed into other_names

```
>>> first_name, *other_names = input("Enter your names
seperated by a space : ").split(" ")
Enter your names seperated by a space : John Jacob Smith
>>> first_name
'John'
>>> other_names
['Jacob', 'Smith']
```

Python – If

An if statement is the fundamental transfer of control statement. If a Boolean expression is True , the if block is executed. If False , the block is skipped. Execution continues immediately after the block

python-if_statement.py

```
#!/usr/bin/env python

theanswer=42
prompt="You are correct!"
_youranswer=input("what is the answer to life? ")
youranswer=int(_youranswer)
if(youranswer!=theanswer):
        print("Really!?")
        prompt="""Please read:
                the Hitchhicker's Guide to the
Galaxy"""
        print(prompt)
```

Python – If Else

The else statement adds a False branch to the if statement. That branch (block) executes if the condition is False .

Here is the syntax:

```
if(Boolean):
    True block
else:
    False block
```

python-if__else_statement.py

```
#!/usr/bin/env python
theanswer=42
_youranswer=input("what is the answer to life? ")
youranswer=int(_youranswer)
if(youranswer!=theanswer):
        print("Really!?")
        prompt="""Please read:
                the Hitchhicker's Guide to the Galaxy"""
else:
        prompt="You are correct!"
print(prompt)
```

Python – Elif

Deep nesting of if statements can lower the readability of code. The elif statement is an alternative syntax.

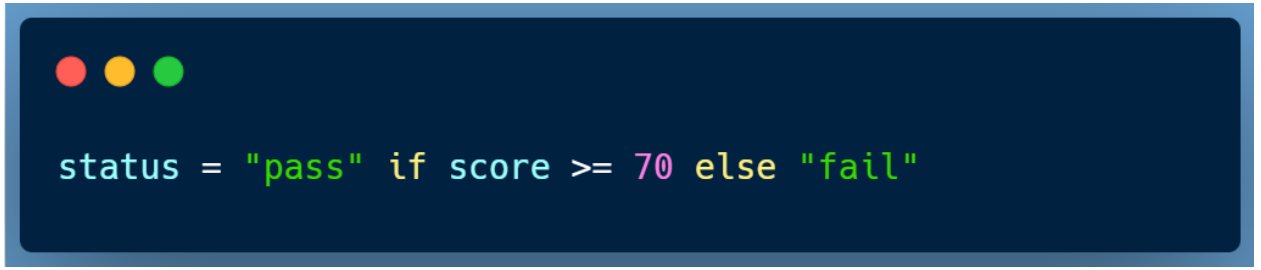
python-elif_statement.py

```
#!/usr/bin/env python
employee=“E"
if employee=="E":
        print("calculate exempt pay")
elif employee=="H":
        print("calculate hourly pay")
elif employee=="M":
        print("calculate management pay")
else:
        pass  # do nothin
```

Python – Ternary Expression

Python supports a **ternary expression** for simple conditional assignments.

It allows you to assign a value based on a condition in a single line. This is useful for short, readable decisions.



Python – Break and continue

- *break* statement: interrupts an iteration. Immediately ends the current loop and proceeds to the next statement after the iteration block.
- *continue* statement: skips the remainder of the current iteration. Resumes at the top of the next iteration.

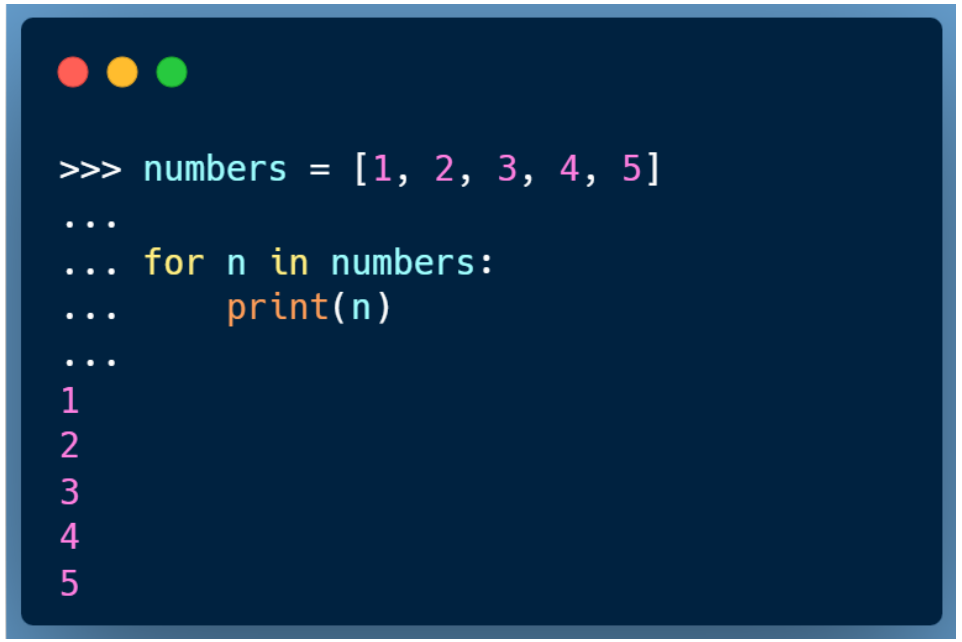
python-break-continue.py

```
while(True):
    var=input("Enter  value:  ")
    if  var:
        continue
    else:
        break
```

Python – For Loop

A **for loop** is used to iterate over a sequence of values, such as a list, range, or string. It executes the same block of code once for each item in the sequence.

for loops are commonly used for iteration, searching, and processing data.



Python – For Loop

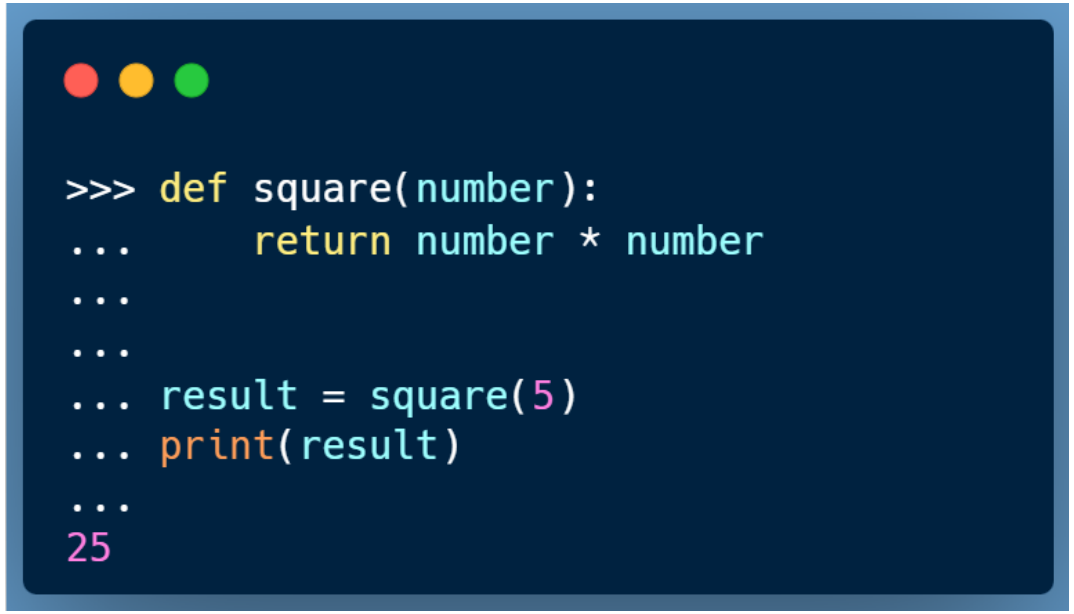
- *break* statement: interrupts an iteration. Immediately ends the current loop and proceeds to the next statement after the iteration block.
- *else:* Only executed if the break is never reached

```python
import random

numbers = random.sample(range(1, 21), 5)
print(numbers)

target = int(input("Enter a number: "))

for n in numbers:
    if n == target:
        print("Number found!")
        break
else:
    print("Number not found.")
```

Python – Functions

A **function** is a named block of code that performs a specific task. Functions allow you to group logic, reuse code, and keep programs organized.

They can accept inputs, perform operations, and return results. Functions make code easier to read, test, and maintain.



Python – Data Structures

Python – Built-in Data Structures

Python includes several **built-in data structures** used to store and organize data.Each structure is designed for different access patterns and use cases.

Common built-in data structures include:

List – ordered, mutable collection
Tuple – ordered, immutable collection
Set – unordered collection of unique values
Dictionary – key–value pairs

Choosing the right data structure makes code simpler, faster, and easier to reason about.

Python – List

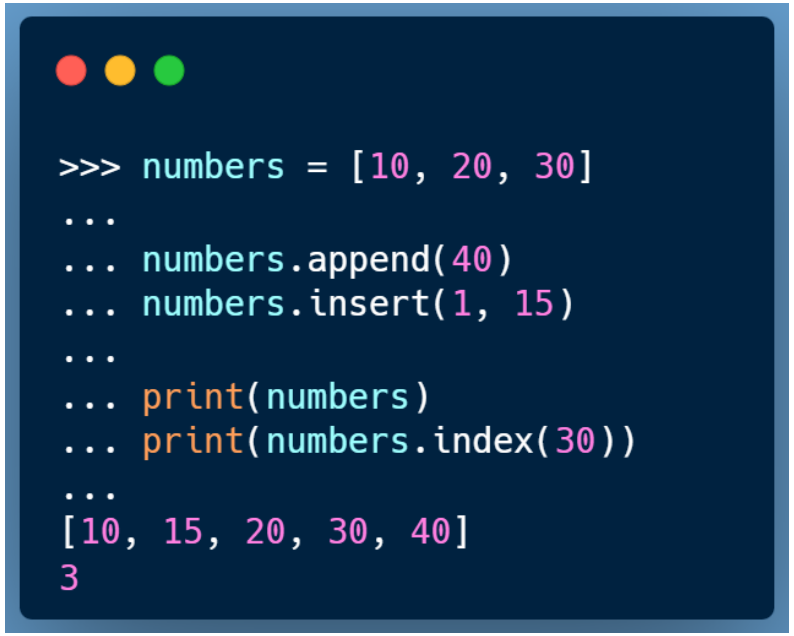
A **list** is an ordered, mutable collection of values.

Use lists when:

- Order matters
- You need to add, remove, or modify items
- You need to iterate sequentially

Avoid lists when:

- You need guaranteed uniqueness
- You want immutability or fixed structure
- Lists are the most flexible and commonly used data structure.

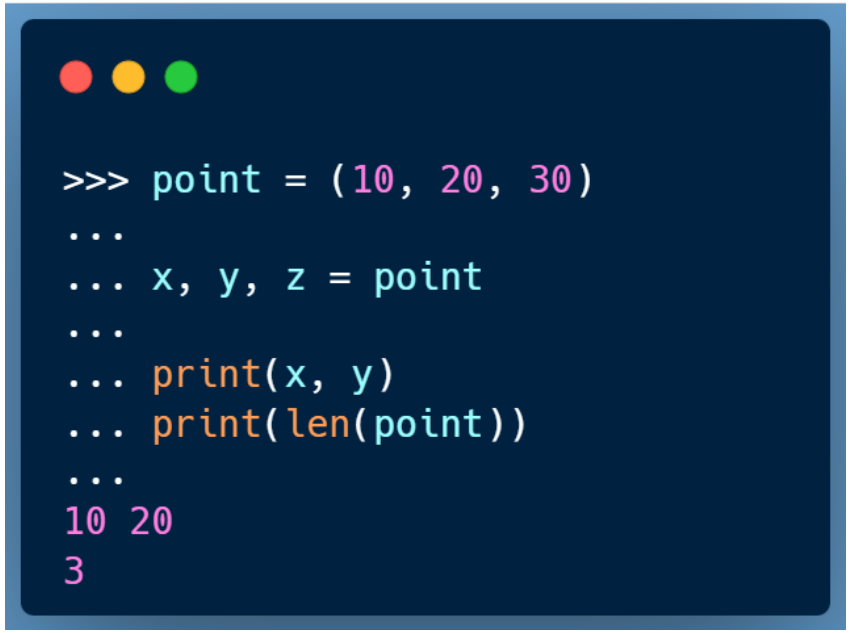


Python – Tuple

A **tuple** is an ordered, immutable collection of values.

Use tuples when:
- The data should not change
- You want to represent a fixed record
- You want safer shared data

Avoid tuples when:
- You need to modify values
- You need dynamic resizing
- Tuples communicate intent: *this data is not meant to change*.



Python – Set

A **set** is an unordered collection of unique values.

Use sets when:
- You need uniqueness
- Order does not matter
- You need fast membership checks

Avoid sets when:
- You care about order
- You need duplicate values

Sets are ideal for filtering, comparison, and de-duplication.

```
>>> values = {1, 2, 3}
...
... values.add(4)
... values.update([3, 5, 6])
...
... print(values)
... print(3 in values)
...
{1, 2, 3, 4, 5, 6}
True
```

Python – Dictionary

A **dictionary** stores data as key–value pairs.

Use dictionaries when:

- You need to associate values with keys
- Fast lookups are important
- Data is naturally named or structured

Avoid dictionaries when:

- Data is purely sequential
- Order or position is the primary concern

Dictionaries are the backbone of configuration, JSON, and structured data in Python.

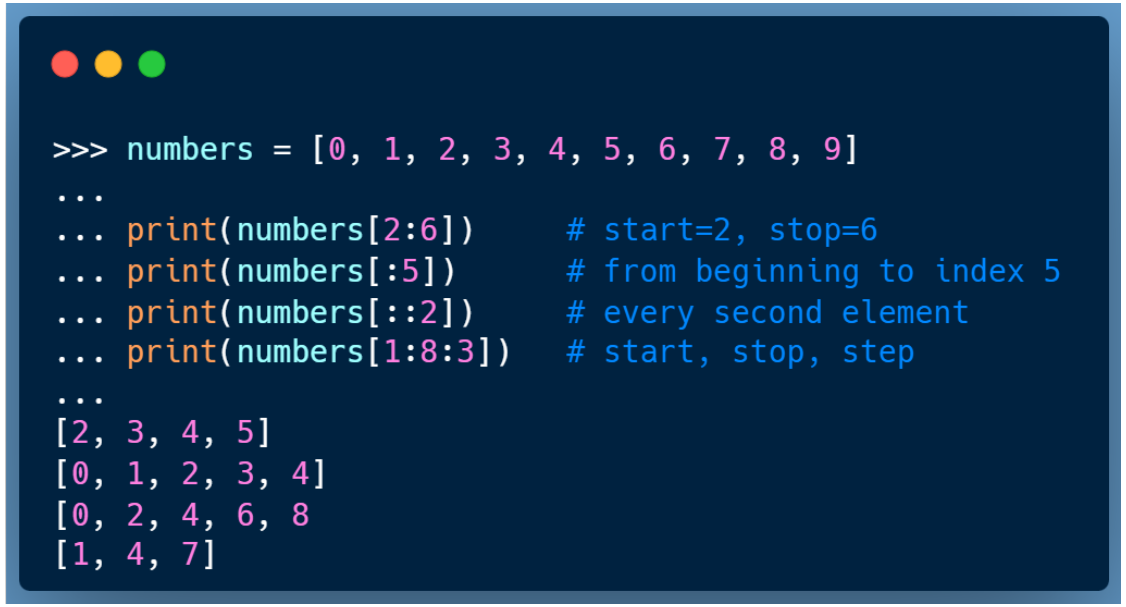
Python – Slicing

Slicing allows you to extract a portion of a sequence such as a list, string, or tuple.

Slices use the format **start:stop:step**.

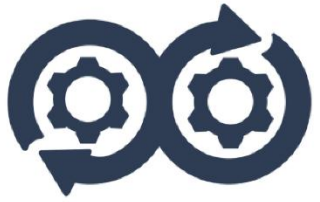
- start – where the slice begins (inclusive)
- stop – where the slice ends (exclusive)
- step – how many items to skip

Slicing creates a new sequence without modifying the original.



```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
...
... print(numbers[2:6])     # start=2, stop=6
... print(numbers[:5])      # from beginning to index 5
... print(numbers[::2])     # every second element
... print(numbers[1:8:3])   # start, stop, step
...
[2, 3, 4, 5]
[0, 1, 2, 3, 4]
[0, 2, 4, 6, 8
[1, 4, 7]
```

Lab: Python Data Structures



POP QUIZ:

What data structure can only store hash-able objects?

A. List
B. Tuple
C. Dictionary
D. Set

POP QUIZ:

What data structure can only store hash-able objects?

A. List
B. Tuple
C. Dictionary
D. **Set**

POP QUIZ:

How do you check if a dictionary contains a specific key?

A. key in dict.values()
B. dict.has(key)
C. key in dict
D. dict.contains(key)

POP QUIZ:

How do you check if a dictionary contains a specific key?

A. key in dict.values()
B. dict.has(key)
C. **key in dict**
D. dict.contains(key)

POP QUIZ:

a = (1, 2, 3, 4, 5); Which assignment correctly inserts 2.5 after index 1?

A. a = a[0:2] + (2.5,) + a[2:]
B. a.insert(2, 2.5)
C. a.append(2.5)
D. a = a[0:3] + (2.5) + a[2:]

Object Oriented Programming

Python – OOP

Object-Oriented Programming is a way of structuring code around **objects** that combine data and behavior into a single unit.

- Objects represent real-world or logical concepts
- Classes act as blueprints for objects
- Data is stored as attributes
- Behavior is defined using methods

OOP helps organize and scale programs as they grow in complexity.

Python – OOP

You can think of a class as a template for a new data type. Everything in Python is an object.

When you create an instance of a class, Python creates an object based on that template.

Python – Class

```python
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species


dog = Animal("Buddy", "Dog")
print(dog.name)
print(dog.species)
```

A **class** is a blueprint used to create objects.
It defines what data an object holds and what actions it can perform.

The __init__ method runs when a new object is created and is used to initialize its state.

- Defines the structure of an object
- Groups data and behavior together
- __init__ sets initial values
- Objects are created from classes

Classes allow you to model real-world concepts in code.

Python – Methods

```
>>> class Animal:
...     def __init__(self, name, species):
...         self.name = name
...         self.species = species
...
...     def speak(self):
...         return f"{self.name} makes a sound"
...
...
... dog = Animal("Buddy", "Dog")
... print(dog.speak())
...
Buddy makes a sound
>>>
```

Methods are functions that belong to a class and operate on an object's data. The **self** parameter represents the **calling object** and is automatically passed as the first argument to every method.

- Methods define object behavior
- self refers to the current instance
- Used to access or modify object data
- Always the first parameter in instance methods

Understanding self is key to working with objects in Python.

Python – Special Methods

```
>>> class Animal:
...     def __init__(self, name, species):
...         self.name = name
...         self.species = species
...
...     def __eq__(self, other):
...         return self.name == other.name \
...                and self.species == other.species
...
...
... a1 = Animal("Buddy", "Dog")
... a2 = Animal("Buddy", "Dog")
...
... print(a1 == a2)
... a1.species = "Cat"
... print(a1 == a2)
...
...
True
False
```

Python provides **special methods** (sometimes called *dunder methods*) such as __init__, __eq__, and others.

You do not call these methods directly. Instead, the Python interpreter invokes them automatically when appropriate.

For example, __eq__ is called when two objects are compared using ==.

- Special methods define object behavior
- Automatically invoked by Python
- Used for comparison, initialization, printing, and more

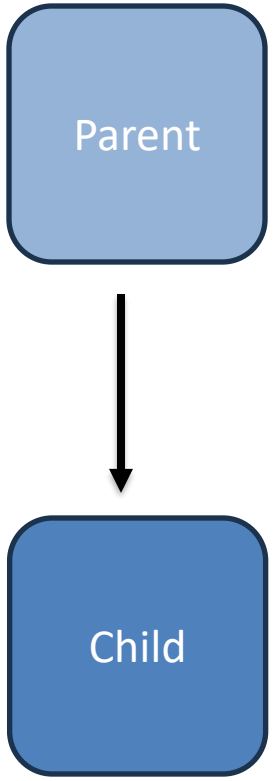
Special methods let your objects behave like built-in types.

Python – Inheritence

A child class **inherits** from a parent class. This means the child automatically has access to the parent's attributes and methods.

The parent defines **shared behavior**, while the child can add or customize its own behavior.

Think of the parent as a general concept and the child as a more specific version of it.



Python – Inheritance

```
>>> class Animal:
...     def __init__(self, species):
...         self.species = species
...
...     def speak(self):
...         return "The animal makes a sound"
...
...
... class Dog(Animal):
...     pass
...
...
... dog = Dog("Dog")
... print(dog.species)
... print(dog.speak())
...
Dog
The animal makes a sound
>>>
```

When a class inherits from another class, it automatically receives the parent's attributes and methods. The child class can use everything defined in the parent without adding any code of its own.

- Parent defines shared data and behavior
- Child receives those features automatically
- No duplication required

Inheritance allows common behavior to be defined once and reused across many classes.

Python – Inheritance

Python does not use access keywords like public, private, or protected. Instead, Python relies on **naming conventions** to communicate intent, not enforcement.

- No underscore → public by convention
- Single underscore (_name) → internal use
- Double underscore (__name) → name mangling (Still accessible, but discouraged)

These conventions are not enforced by the language, but respecting them is considered **Pythonic** and leads to clearer, more maintainable code.

Python – Overriding

```
>>> class Animal:
...     def speak(self):
...         return "The animal makes a sound"
...
...
... class Dog(Animal):
...     def speak(self):
...         return "Bark"
...
...
... animal = Animal()
... dog = Dog()
...
... print(animal.speak())
... print(dog.speak())
...
The animal makes a sound
Bark
```

Overriding occurs when a child class provides its own implementation of a method defined in the parent class. The child's method replaces the parent's version when called on the child object.

- Uses the same method name
- Allows specialized behavior
- Works with inheritance and polymorphism

Overriding lets child classes customize behavior while keeping a common interface.

Python – Overloading

```
>>> class Animal:
...     def speak(self, *args):
...         if not args:
...             print("The animal is silent")
...         else:
...             for arg in args:
...                 print(arg)
...
...
... a = Animal()
... a.speak()
... a.speak(1, 2, 3)
... a.speak("End!")
...
The animal is silent
1
2
3
End!
```

Python does not support function overloading by defining multiple functions with the same name. Similar behavior can be achieved by using **flexible parameters**, such as *args.

This allows a single function to accept different numbers of arguments and handle them at runtime.

Python – Super()

The **super()** keyword is used to call methods from a parent class.It allows a child class to reuse parent behavior without hard-coding the parent's name.

In some cases, you can also call the parent method using the class name directly, but super() is preferred because it works correctly with inheritance chains.
Use super() when extending behavior instead of rewriting it.

Python

```
>>> class Animal:
...     def speak(self):
...         return "The animal makes a sound"
...
...
... class Dog(Animal):
...     def bark(self):
...         return super().speak() + " and
barks"
...
...
... dog = Dog()
... print(dog.bark())
...
The animal makes a sound and barks
>>>
```

A child class can define **new methods** while still using behavior from the parent class. The child does not override the parent method, but **calls it from its own method**.

- Parent defines shared behavior
- Child adds new behavior
- Uses super() to reuse logic

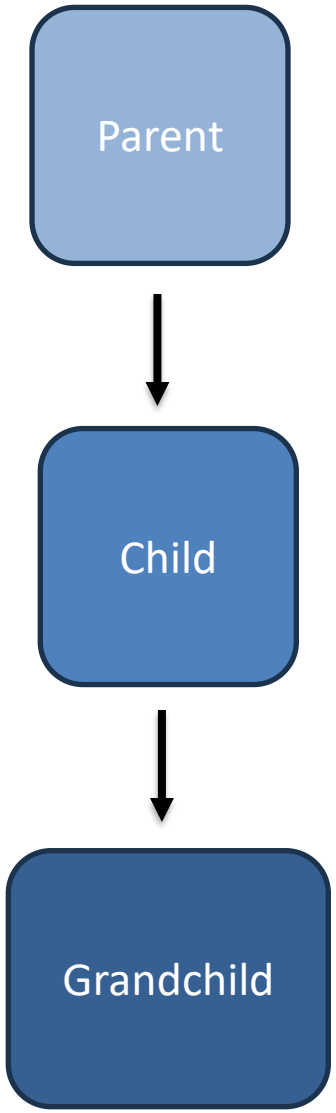
This avoids duplication while keeping responsibilities clear.

Python – Grandchildren

Now add a **child of the child** (a grandchild of the original parent). Inheritance can continue through multiple levels.

Each level becomes more **specialized**, while still retaining everything above it. Behavior flows **downward** through the hierarchy unless it is overridden.

As you go down the tree, objects become more specific.



Python – Overriding

```
>>> class Animal:
...     def speak(self):
...         return "The animal makes a sound"
...
...
... class Feline(Animal):
...     def speak(self):
...         return "The feline makes a sound"
...
...
... class Cat(Feline):
...     def speak(self):
...         return Animal.speak(self) + " and meows"
...
...
... cat = Cat()
... print(cat.speak())
...
The animal makes a sound and meows
>>>
```

A child class can **override a method** and still explicitly choose **which parent's implementation** to call. This allows precise control over behavior in deeper inheritance hierarchies.

- Methods can be overridden at any level
- super() follows the inheritance chain
- Parent classes can also be called directly if needed

This gives developers flexibility while keeping inheritance explicit.

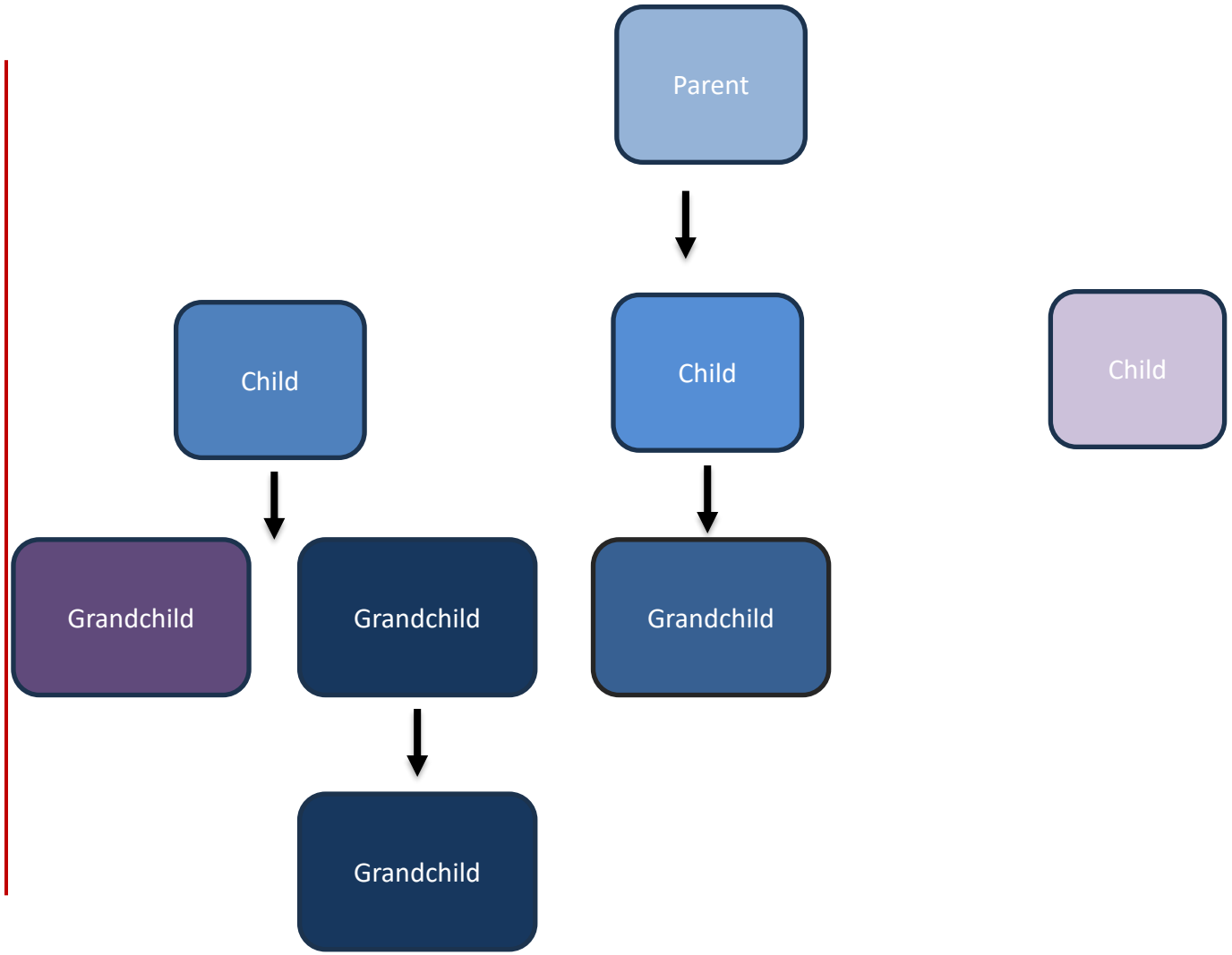
Python – Grandchildren

Now follow a single branch downward:

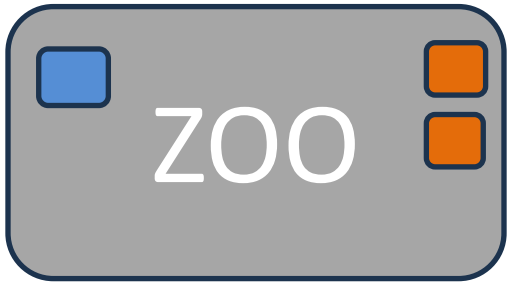
Parent → Child → Grandchild → Grandchild

Each generation becomes **more specialized**, but never loses what came before it. Behavior flows **downward** unless overridden.

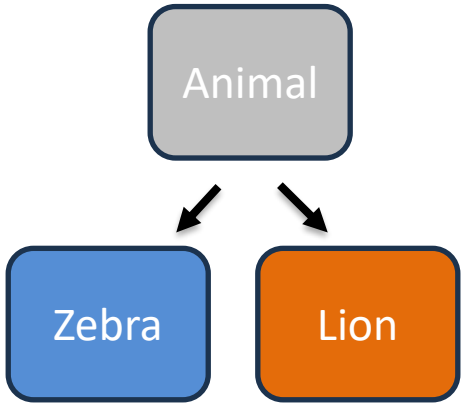
This is why inheritance trees are often narrow at the top and wider as you go down.



Python – Polymorphism



"Zoo" can handle anything that inherits from Animal



You can make your own custom class long as it extends Animal

Even though the objects are **different shapes at the bottom**, they all trace back to the same **Parent**.

That means:
Code written to handle the **Parent**
Can also handle **any Child or Grandchild**
Without knowing which one it is.

You can say:
"As long as something is a Parent, I don't care which Child it actually is."
That is **polymorphism**.

Python – Abstraction

Abstraction is the idea of exposing **what an object does** while hiding **how it does it**.

It allows code to depend on **interfaces and behavior**, rather than concrete implementations.

- Focuses on *what* an object can do
- Hides internal implementation details
- Encourages loose coupling
- Works naturally with inheritance and polymorphism

Abstraction helps you write code that is easier to extend, change, and reason about.

Python – Abstraction

```
>>> from abc import ABC, abstractmethod
...
... class Animal(ABC):
...     @abstractmethod
...     def speak(self):
...         pass
...
...
... class Lion(Animal):
...     def speak(self):
...         return "Roar"
...
...
... lion = Lion()
... print(lion.speak())
...
Roar
```

An **abstract class** defines **what a child class must do**, without providing the implementation.

It acts as a contract that child classes are required to follow.

- Declares required methods
- Does not implement behavior
- Forces children to provide implementations
- Works with inheritance and polymorphism

Abstraction ensures consistency while allowing flexibility in behavior.

Python – Polymorphism and Abstraction

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def feed(self):
        pass


class Zoo:
    def feed_animal(self, animal: Animal):
        animal.feed()
```

```python
from zoo import Animal, Zoo

class Lion(Animal):
    def feed(self):
        print("Feeding the lion")


zoo = Zoo()
lion = Lion()

zoo.feed_animal(lion)
```

Zoo doesn't care *what* animal it gets. It only cares that the animal knows how to feed itself.

This means the developers of Zoo can assume users who make animals will fulfil certain requirements.

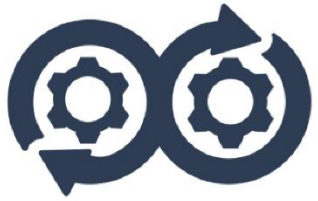
Python – OOP Whats Next

We covered the core OOP concepts needed to read and write Python effectively. There are additional OOP-related tools and patterns commonly used in real-world Python applications.

- Static methods and class methods
- Getters and setters (accessors/mutators)
- @dataclass
- Pydantic models
- Object-Relational Mappers (ORMs)

These topics build on the same OOP foundations and are best explored once the basics are solid.

Lab: Python OOP



POP QUIZ:

What concept allows children to be treated as their parents?

A. Polymorphism
B. Inheritance
C. Abstraction
D. Multi-Inheritance

POP QUIZ:

What concept allows children to be treated as their parents?

A. **Polymorphism**
B. Inheritance
C. Abstraction
D. Multi-Inheritance

POP QUIZ:

What must a child class do with an abstract function?

A. Extend it
B. Overload it
C. Implement it
D. Pass it

POP QUIZ:

What must a child class do with an abstract function?

A. Extend it
B. Overload it
C. **Implement it**
D. Pass it

POP QUIZ:

What does implementing the __lt__ function do?

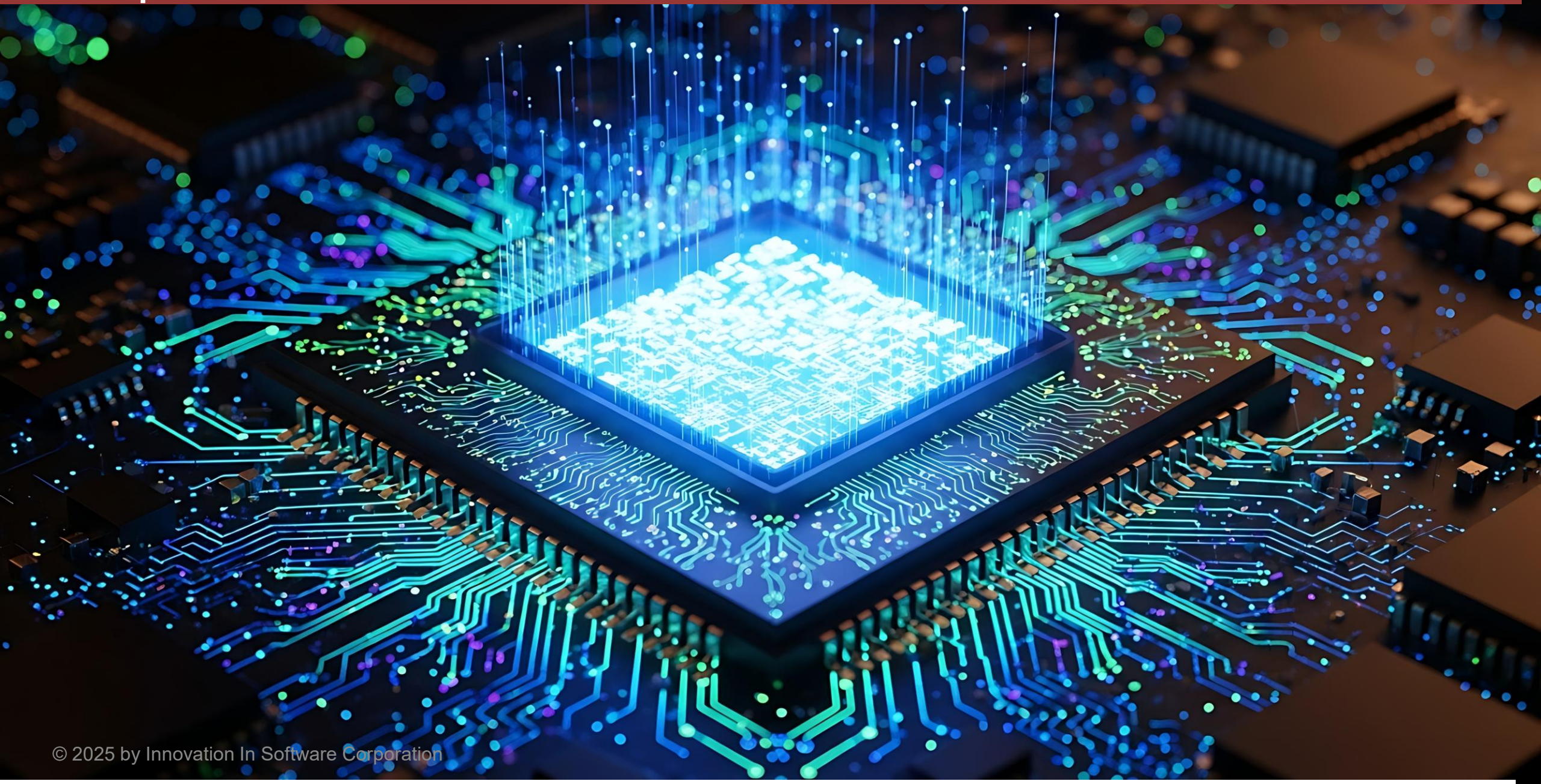
A. Makes the class light
B. Adds metadata about the class
C. Makes the object less than
D. Allows you to use the less than operator on a class instance

POP QUIZ:

What does implementing the __lt__ function do?

A. Makes the class light
B. Adds metadata about the class
C. Makes the object less than other objects that you compare it to
D. **Allows you to use the less than operator on a class instance**

Comprehensions, Generators, Decorators



Python – List Comprehension

```
>>> numbers = [3, 7, 12, 18, 5, 22]
... result = []
...
... for n in numbers:
...     if n > 10:
...         result.append(n)
...
... print(result)
...
[12, 18, 22]
```

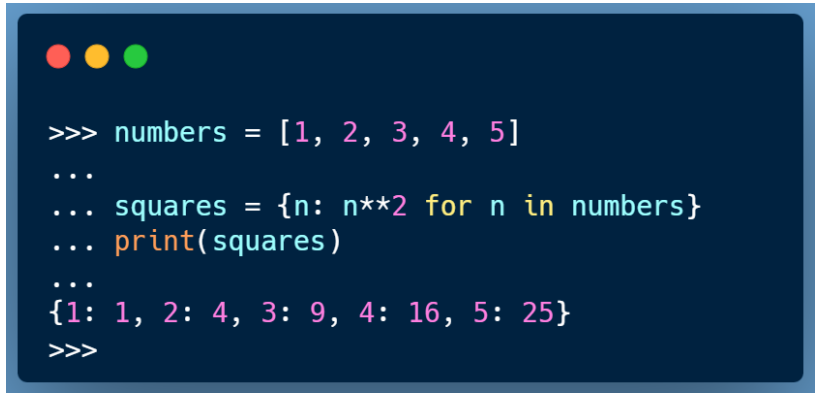
```
>>> numbers = [3, 7, 12, 18, 5, 22]
...
... result = [n for n in numbers if n > 10]
... print(result)
...
[12, 18, 22]
```

A **list comprehension** is a concise way to create a new list from an existing sequence. It combines iteration and optional filtering into a single, readable expression.

- Creates lists in a compact form
- Can include conditions
- Often clearer than a full loop
- Common in Python code

List comprehensions are a powerful tool for transforming data in Python.

Python – More Comprehension



Python also supports comprehensions for **dictionaries** and **sets**, using the same pattern as list comprehensions. They allow you to create new collections concisely while transforming or filtering data.

- Dictionary comprehensions create key–value pairs
- Set comprehensions create unique values
- Both support conditions and expressions

These comprehensions help express data transformations clearly and efficiently.

Python – Nested Comprehension

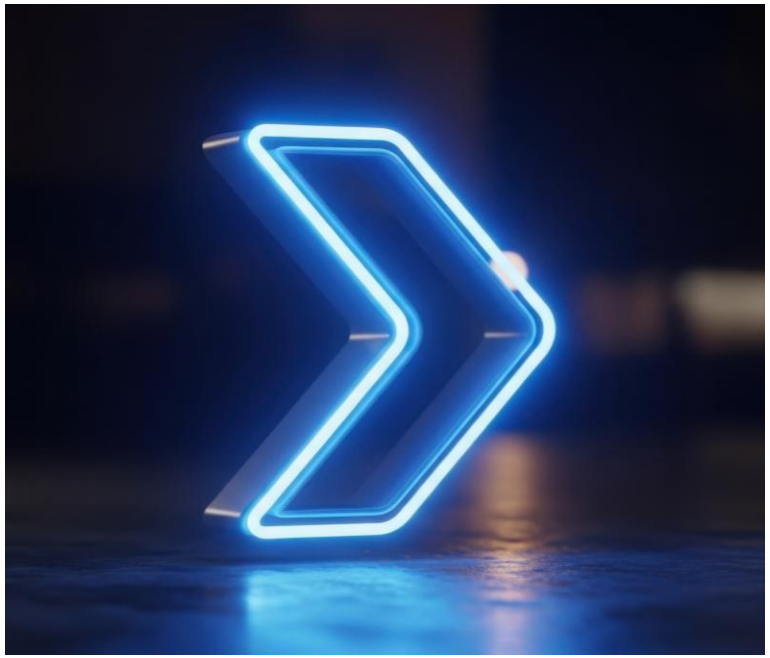
```
>>> matrix = [
...     [1, 2, 3],
...     [4, 5, 6],
...     [7, 8, 9]
... ]
...
... flattened = [
...   num for row in matrix
...           for num in row
... ]
... print(flattened)
...
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A **nested list comprehension** is a list comprehension inside another list comprehension. It is commonly used to flatten data or work with nested structures.

- Represents nested loops in a compact form
- Useful for transforming nested data
- Should stay readable and simple

If a nested comprehension becomes hard to read, a regular loop is often better.

Python – Generators



A **generator** is a special kind of function that produces values **one at a time** instead of all at once. Generators use the yield keyword and remember their state between calls.

- Produce values lazily
- Use less memory than lists
- Ideal for large or infinite sequences

Generators allow you to work with data efficiently as it is needed.

Python – Generators

Calling this function does not return all values immediately. Each call to yield returns the next value and pauses execution until the next iteration.

This allows the function to generate values on demand instead of storing them all in memory.

```
>>> def count_up_to(n):
...     for i in range(1, n + 1):
...         yield i
...
>>> nums = count_up_to(3)
>>> next(nums)
1
>>> num2  = next(nums)
>>> print(num2)
2
>>> next(nums)
3
>>> next(nums)
Traceback (most recent call last):
  File "<python-input-28>", line 1, in <module>
    next(nums)
    ~~~~^^^^^^
StopIteration
>>> for i in count_up_to(4):
...     print(i)
...
1
2
3
4
>>>
```

Python – Decorators

Decorators are applied **above functions or class definitions** to modify their behavior. A decorator is simply a **function**. The object being decorated is passed into the decorator as an argument.

The decorator returns a function, usually the same one, but **enhanced with additional behavior**. Decorators allow behavior to be added without modifying the original code.

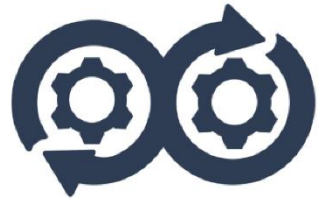
Python – Decorators

This **timer decorator** measures how long a function takes to run. It wraps the original function, runs it, and adds timing logic before and after execution.

This is a common use of decorators for profiling and debugging. Decorators allow this behavior without changing the original function's code.

```
>>> import time
...
... def timer(func):
...     def wrapper(*args, **kwargs):
...         start = time.time()
...         result = func(*args, **kwargs)
...         end = time.time()
...         print(f"{func.__name__} took {end - start:.4f} seconds")
...         return result
...     return wrapper
...
...
... @timer
... def slow_function():
...     time.sleep(1)
...
...
... slow_function()
...
slow_function took 1.0012 seconds
>>>
```

Lab: Python Comprehensions, Generators, Decorators



POP QUIZ:

Which comprehension returns a list of numbers **greater than 10**?

A. [n > 10 for n in nums]
B. [n for n in nums if n > 10]
C. [nums if n > 10 for n in nums]
D. [n for n > 10 in nums]

POP QUIZ:

Which comprehension returns a list of numbers **greater than 10**?

A. [n > 10 for n in nums]
B. **[n for n in nums if n > 10]**
C. [nums if n > 10 for n in nums]
D. [n for n > 10 in nums]

POP QUIZ:

Which statement about generators is true?

A. Generators store all values in memory
B. Generators must return a list
C. Generators produce values lazily
D. Generators cannot be iterated

POP QUIZ:

Which statement about generators is true?

A. Generators store all values in memory
B. Generators must return a list
C. **Generators produce values lazily**
D. Generators cannot be iterated

POP QUIZ:

What is the purpose of a decorator?

A. To replace a function entirely
B. To modify behavior without changing the function code
C. To execute a function immediately
D. To declare a function private

POP QUIZ:

What is the purpose of a decorator?

A. To replace a function entirely
B. **To modify behavior without changing the function code**
C. To execute a function immediately
D. To declare a function private

Congratulations

Congratulations — you've covered the foundations of Python programming.
You learned how to work with core data structures, understand and use basic Object-Oriented Programming concepts, and apply Pythonic patterns such as generators, comprehensions, and decorators.

These fundamentals form the base for writing clean, readable, and scalable Python code.
Great work — you're now ready to build on this foundation.