HO CHI MINH CITY UNIVERSITY OF TRANSPORT
FACULTY OF INFORMATION TECHNOLOGY
SOFTWARE ENGINEERING DEPARTMENT

CHAPTER 5
ELEMENTARY PROGRAMMING

CONTENTS

Writing a Simple Program

- **Problem**: Calculating the area of a circle.
- **Algorithm:**
  1. Get the circle's radius from the user.
  2. Compute the area by applying the following formula:
     area = radius * radius * PI
  3. Display the result.
- **Important issues**:
  - How to read the radius?
  - How to store the radius, area in the program?
    - Values for the radius and area are stored in the computer's memory
  - How to display result?

Writing a Simple Program

- **Program**

```
# Assign a value to radius
radius = 20
# Compute area
area = radius * radius * 3.14159
 # Display results
print("The area for the circle of radius", radius, "is", area)
```

```
The area for the circle of radius 20 is 1256.636
```

Keywords

- **Keywords** that are reserved words, have special meanings. They cannot be used as variable names, function names, or any other identifiers.

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

Identifier

- **Identifier** is the name that identify the element in a program such as variable, function, array….
- All identifiers must obey the following **rules**:
  - Consists of letters, digits, and underscores (_)
  - Must start with a letter or an underscore, cannot start with a digit
  - Cannot be a keyword
  - Can be of any length
- **Example**:
  - Legal identifiers: area, radius, top_of_page, temp1…
  - Not legal identifiers: 2A, for, search elem

Identifier

- **Note**:
  - Python is ***case sensitive*** → area, Area, and AREA are all different identifiers
- **Tip**:
  - Descriptive identifiers → easy to read, understand, maintain
  - Use lowercase letters for variable names
  - Two major ways to paste words together:
    - Underscore: radius_of_circle
    - Camel Case: radiusOfCircle
- **Quiz**:
  - Which of the following identifiers are valid?

    miles, Test , a+b, b–a, 4#R, $4, #44, if

Data Types

- A data type consists of two things:
  - A set of values it can have
  - A set of operations that can be performed on those values
- ***Basic data types***:
  - Integer
  - Floating-Point Number
  - Complex Number
  - String
  - Boolean

Data Types

1. Integer

- Contains values: -3, 3, 0, -1, 1234,…
- Type: **int**
- An integer value in other bases

Prefix	Base
0b (zero + 'b') 0B (zero + 'B')	2
0o (zero + 'o') 0O (zero + 'O')	8
0x (zero + 'x') 0X (zero + X')	16

```
type(2)
```

```
int
```

```
type(0b1010)
```

```
int
```

```
type(0o56)
```

```
int
```

```
type(0x12A)
```

```
int
```

Data Types

2. Floating-Point Number

- Contains real values with a decimal point: -3.5, 3.1419, 4.2e-4…
- Type: **float**
- *Precision*: float values are only approximations to real numbers.

```
type(3.5)
```

```
float
```

```
2/3 + 1
```

```
1.6666666666666665
```

```
4.2e-4
```

```
0.00042
```

```
type(1/4)
```

```
float
```

```
5/3
```

```
1.6666666666666667
```

Data Types

3. Complex Number

- Complex numbers are specified as <real part> + <imaginary part>j
- Type: **complex**

```
In [37]: 2 + 3j

Out[37]: (2+3j)

In [36]: type(2 + 3j)

Out[36]: complex
```

Data Types

4. String

- String is sequence of character data.
- Type: **str**
- String literals may be delimited using either single or double quotes
  - ‘Hello’
  - “Hello”
- A string can also be empty: ‘’
- To include special character in a string → using a backslash (\)

```
In [39]: print("Hello")
Hello

In [40]: print('Hello')
Hello

In [41]: type("Hello")
Out[41]: str
```

```
In [42]: print('This string contains a single quote (') character.')
  File "<ipython-input-42-4fb72c6c5730>", line 1
    print('This string contains a single quote (') character.')
                                                  ^
SyntaxError: invalid syntax
```

Data Types

4. String

Special Character

Escape Sequence	Example
\'	In [43]: `print('This string contains a single quote (\') character.')` `This string contains a single quote (') character.`
\"	In [44]: `print("This string contains a double quote (\") character.")` `This string contains a double quote (") character.`

\newline	In [46]: `print("Hello!\`
	` How are you?")`
	`Hello! How are you?`

\\	In [48]: `print("This is backslash(\\)")` `This is backslash(\)`

Data Types

4. String

Applying Special Meaning to Characters

Escape Sequence	Interpretation
\t	Horizontal Tab (TAB) character
\n	New line
\a	Bell (BEL) character
\b	Backspace character
\r	Carriage Return (CR) character

Data Types

4. String

Example:

```
In [50]: print("Name\tAge")

Name    Age

In [51]: print("Name\nAge")

Name
Age

In [52]: print("Name\aAge")

Name•Age

In [53]: print("Name\bAge")

NamAge

In [70]: print("YourName\rAge")

AgerName
```

Data Types

5. Boolean

- Boolean type may have one of two values, True (1) or False (0)
- Type: **bool**
- Expressions are often evaluated in Boolean context.

```
In [71]: 5 > 4
Out[71]: True

In [72]: 5 < 4
Out[72]: False

In [73]: type(5 < 4)
Out[73]: bool
```

Variables

- **Variables** are the names that refers to *different* values stored in memory. Every value has a data type.
- Naming variables must follow the rules of identifiers.
- Variable Creation
  - Assign values to a variable

    variable = expression
  - *'='*: *gets, not equals*
- We can reassign the value of a variable

```
In [9]: radius = 1.0 # Assign 1.0 to variable radius, radius is created
        print(radius)
        y = radius #Assign value of radius to y
        area = radius*radius*3.14159 # radius is used
        radius = 1.5
        print(radius)
```

```
1.0
1.5
```

Variables

- **NOTE**:

A variable must be created before it can be used.

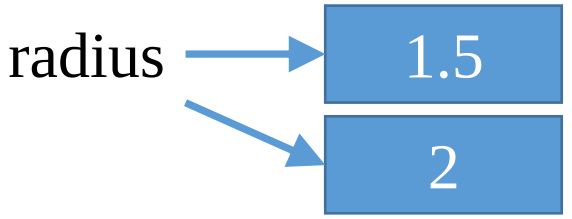
```
In [10]: count = count + 1

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-cd3c5d6dc43e> in <module>
----> 1 count = count + 1

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

Must place the *variable name to the left of the assignment operator*

```
In [11]: 1 = count
  File "<ipython-input-11-197f8d923747>",
    1 = count
      ^
SyntaxError: can't assign to literal
```

Variables

- **NOTE**:
  Python is a dynamically typed language



Dynamic Typing

- Interpreter does type checking only as code runs
- Type of a variable is allowed to change over its lifetime

Static Typing

- Type checking is performed as your program is compiled
- A variable generally is not allowed to change type.

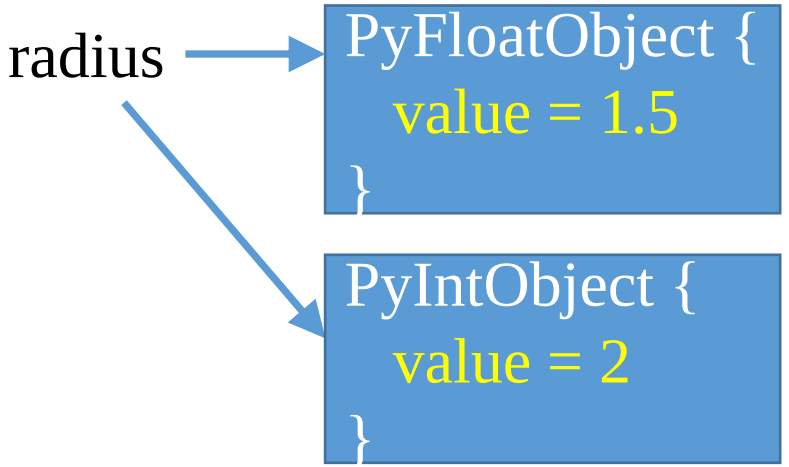
```
In [6]: radius = 1.5
        print(radius)
        type(radius)

        1.5

Out[6]: float
```

```
In [7]: radius = 2
        print(radius)
        type(radius)

        2

Out[7]: int
```

Variables

Object References

- Python is a highly object-oriented language → every item of data in a Python program is called an **object**.
- Objects can be very small (the number 3) or very large (a digital photograph)
- Every object stored in memory location.

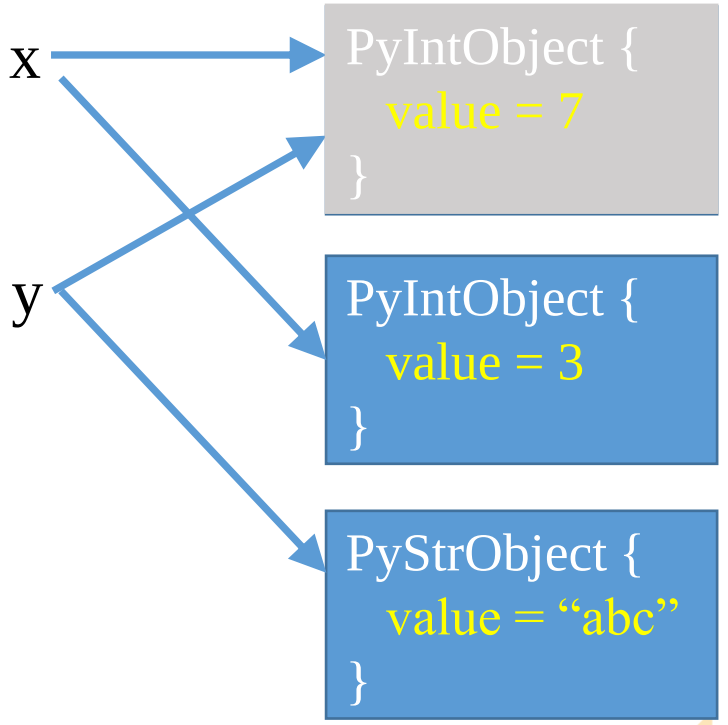


Variables

Object References

```
In [5]: #create an int object with value 7
        #and new variable x
        x = 7
        #create a new variable y
        y = x
        print(x,", ",y)
        #create a new int object with value 3
        x = 3
        print(x,", ",y)
        #create a new str object with value "abc"
        y = "abc"
        print(x,", ",y)
```

```
7 ,  7
3 ,  7
3 ,  abc
```



- There is no longer any reference to the integer object with value 7. What happens with this object???

Variables

- **NOTE:**

Lifetime of an object

- An object's life begins when it is created, at which time at least one reference to it is created.
- When there is no reference to object:
  → lifetime is over
  → it is inaccessible
  → reclaim the allocated memory (*garbage collection*)

Assignment Statement

- The general syntax:

  variable = expression

- This is executed as follows:
  1. Evaluate the expression to produce a value.
  2. Assign value of expression to variable.
- An **expression** represents a computation involving *values*, *variables*, *and operators* that, taken together, evaluate to a value.

```
In [12]: import math
         y = 2
         x = 1
         x = x + 1
         x = y
         x = math.sqrt(5)
         print(x)

         2.23606797749979
```

Assignment Statement

- Syntax to assgin a value to multiple variables :

  var1 = var2 =…varn = expression

- Example:

  i = j = k = 1

  equivalent to

  k = 1

  j = k

  i = j

Assignment Statement

- Syntax of **Simultaneous Assignment**:

var1, var2, ..., varn = exp1, exp2, ..., expn

- Evaluate all the expressions on the right and assign them to the corresponding variable on the left simultaneously.

Using temporary variable

```
1  a = 1
2  b = 2
3  print("a =",a,", b =",b)
4  #swap a and b
5  temp = a # Save a in temp variable
6  a = b    # Assign the value in b to a
7  b = temp # Assign the value in temp to b
8  print("After swapping")
9  print("a =",a,", b =",b)
```

```
a = 1 , b = 2
After swapping
a = 2 , b = 1
```

Using simultaneous assignment

```
1  a = 1
2  b = 2
3  print("a =",a,", b =",b)
4  #swap a and b
5  a, b = b, a
6  print("After swapping")
7  print("a =",a,", b =",b)
```

```
a = 1 , b = 2
After swapping
a = 2 , b = 1
```

Assignment Statement

- **Simultaneous Assignment**

Example: Input three numbers and obtains their average.

In [18]:
```
# Prompt the user to enter three numbers
num1, num2, num3 = eval(input(
    "Enter three numbers separated by commas: "))
# Compute average
average = (num1 + num2 + num3) / 3
# Display result
print("The average of", num1, num2, num3,"is", average)
```

```
Enter three numbers separated by commas: 2,3,4
The average of 2 3 4 is 3.0
```

Constants

- A **constant** is an identifier that represents a permanent value (never change).
- Python does not have a special syntax for constants
  →Create a variable to denote a constant.
  →To distinguish a constant from a variable, use all uppercase letters to name a constant
- Example: **PI = 3.14159**
- **Benefits of using constants**:
  1. Don't have to repeatedly type the same value if it is used multiple times.
  2. If you have to change the constant's value, only need to change it once.
  3. Descriptive names make the program easy to read.

Expressions

- An **expression** is a combination of zero or more operators and one or more operands.
- **Operands**: constants, variables, function calls.
- **Operators**: are special tokens that represent computations.
- The interpreter evaluates expression → produces a value.
- **Example**:
  - 42, n, print(“hello”)
  - 3*((i % 4)*(5 +(j – 2)/(k+3)))
  - x >= y
  - x >= 0 and y >= 0

Operators

- **Operators** are used to perform operations on variables and values.
- Python divides the operators in the following groups:
  - Arithmetic operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators
  - Assignment operators

Arithmetic operators

Arithmetic operators are used to perform common mathematical operations:

Operator	Meaning	Types of Operands	Example
+	Addition	int, float, complex, str, bool	9 + 4 → 13 , 9 + 4.5 → 13.5 (2 + 3j) + (3 + 3j) →(5+6j) 'yo' + 'u' → 'you' True + False → 1 True + True → 2
-	Subtraction	int, float, complex, bool	9 - 4 → 5 , 9 - 4.5 → 4.5 (3 +3j) – (2+3j) → (1 +0j) True - False → 1
*	Multiplication	int, float, complex, str, bool	9*4.5 → 40.5 (2 + 3j) * (3 + 3j) → (-3+15j) (2 + 3j) *9 → (18+27j) True * False → 0 'you' * 3 → 'youyouyou'

Arithmetic operators

Operator	Meaning	Types of Operands	Example
/	Float Division	int, float, complex	9 / 4 → 2.25, 9 / 4.5 → 2.0 (1 + 2j) / (4 + 2j) → (0.4+0.3j)
//	Integer Division	int, float	9 // 4 → 2 9 // 4.5 → 2.0
%	Modulus: returns the remainder	int, float	9 % 4 → 1 9.5 % 4 → 1.5
**	Exponentiation	int, float, complex	4 ** 0.5 → 2.0 4.5 ** 3 → 91.125 (1+2j) ** 2 → (-3+4j)

Comparison Operators

- Comparison operators are used to compare two values.
- It either returns **True** or **False**

Operator	Syntax	Meaning	Types of Operands	Example
==	a == b	Equal	int, float, complex, str	1 == 2 → False "he" == "He" → False **1.1 + 2.2 == 3.3 →False**
!=	a != b	Not equal	int, float, complex, str	3 != 3.0 → False (1+2j) != (4+2j) → True
>	a > b	Greater than	int, float, str	7 > 5 → True
<	a < b	Less than	int, float, str	'he' < 'an‘ → False
>=	a >= b	Greater than or equal to	int, float, str	100 >= 100 → True
<=	a <= b	Less than or equal to	int, float, str	100 <= 50 → False

Logical Operators

- Logical operators are used to combine comparison expression.
- Type of Operands is **bool**.
- It either returns **True** or **False**

Operator	Syntax	Meaning	Example
and	a and b	True if both a and b are True False otherwise	((9/3 == 3) and (2*3 ==6)) → True (('A'== 'a') and (3==3)) → False
or	a or b	True if either a or b is True False otherwise	((2==3) or ('A'=='A')) → True
not	not a	True if x is False False if x is True	not(3 == 3) → False

Logical Operators

NOTE: Evaluation of **Non-Boolean** Operands

- *Numeric Value*
  - A zero value is False.
    A non-zero value is True.
- *String*
  - An empty string is False.
    A non-empty string is True.
- *Composite Object*: list, tuple, dict, and set
  - False if it is empty and True if it is non-empty
- *"None" Keyword*

Logical Operators

NOTE: Evaluation of **Non-Boolean** Operands

a	b	a and b	a or b	not a
True	True/False	b	a	False
False	True/False	a	b	True

Compound	Syntax	Meaning	Result
or	x1 or x2 or … xn	True if any of the xi are True	- First xi is True - xn otherwise
and	x1 and x2 and … xn	True if all the xi are True	- xn if all the xi are True - First xi is False

Identity Operators

- Identity operators are used to compare the objects
- not if they are equal, but if they are actually the same object, with the same memory location

Operator	Syntax	Meaning
is	a is b	Returns true if both variables are the same object
is not	a is not b	Returns true if both variables are not the same object

```
In [198]: x = ["apple", "banana"]
          y = ["apple", "banana"]
          z = x
          print(x is y)
          print(z is x)

          False
          True
```

Membership Operators

- Membership operators are used to test whether a value or variable is in a sequence.

Operator	Syntax	Meaning
In	a in b	Returns True if a sequence with the specified value is present in the object
not in	a not in b	Returns True if a sequence with the specified value is not present in the object

```
In [3]: ds = [6,1,4,0]
        print(4 in ds)
        print(5 in ds)
        print(7 not in ds)
```

```
True
False
True
```

Bitwise Operators

- Bitwise operators are used to compare (binary) numbers.

Operator	Syntax	Meaning	Example
&	a & b	Bitwise AND	print(10 & 4) → 0
|	a | b	Bitwise OR	print(10 | 4) → 14
^	a ^ b	Bitwise XOR	print(10 ^ 4) → 14
~	~a	Bitwise NOT	print(~10) → -11
>>	a >> b	Bitwise right shift	print(10 >> 2) → 2
<<	a << b	Bitwise left shift	print(10<<2) → 40

Assignment operators

Assignment operators are used to assign values to variables:

Operator	Syntax	Equivalent
=	a = b	a = b
+=	a += b	a = a + b
-=	a -= b	a = a – b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
//=	a //= b	a = a // b
**=	a **= b	a = a ** b

Operator	Syntax	Equivalent
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b
>>=	a >>= b	a = a >> b
<<=	a <<= b	a = a << b

Operator Precedence

Order	Operator	Order	Operator
1	or	9	+, -
2	and	10	*, /, //, %
3	not x	11	+x, -x, ~x
4	in, not in, is, is not, <, <=, >, >=, !=, ==	12	**
5	|	13	x[index], x[index:index], x(arguments...), x.attribute
6	^	14	(expressions...), [expressions...], {key: value...}, {expressions...}
7	&		
8	<<, >>		

Note: 1 - lowest precedence , 14 - highest precedence

Data Type Conversion

- Data Type Conversion can happen in two ways:
  - Implicit
  - Explicit (*type casting)*

    Required_data_type (expression)

In [4]:
```
#Implicit Data Type Conversion
a = 1
b = 1.5
c = a + b
print(c)
type(c)
```

```
2.5
```

Out[4]:
```
float
```

In [9]:
```
#Explicit Data Type Conversion
x = int(1.5)
print('x = ',x)
y = int('123')
print('y = ',y)
z = float(10)/3
print('z = ',z)
s = str('123')
print('s = ',s) # s = '123'
```

```
x =  1
y =  123
z =  3.3333333333333335
s =  123
```

Statements

- Programs are made up of statements, or instructions
- **Statements** are divided into 2 types:
  - *Simple statements*
    - Function calls
    - Assignment statements
    - Statements: break, continue, return
    - …
  - *Compound statements*
    - Contain other statements
    - Affect or control the execution of those other statements
    - Such as: if , for, while, try, with,…

Indentation

- **Block** of statements:
  - Grouping of statements for a specific purpose
  - Most of the programming languages like C/C++, Java use braces { } to define a block of statements
  - Python uses **indentation** to highlight the block
- Whitespace is used for indentation
  - A block starts with indentation and ends with the first unindented line
  - All statements with the same distance to the right belong to the same block

```
a = 1
b = 2
if a > b:
    print(a," is greater than", b)
else:
    print(a," is not greater than", b)
```

```
1  is not greater than 2
```

Comments

- ***Comments*** *can be used to*:
  - Explain Python code that may not be easy to understand
  - Make the code more readable
  - Prevent execution when testing code.
- Comments are not compiled and executed
- *Creating a* ***Comment:***
  - Using # for single line
  - Using triple quotes (multiline string) for multi-line

Comments

In [10]:
```
#This is my first script
phrase="Hello, world."
print(phrase) #This line displays "Hello,world"
```

```
Hello, world.
```

In [11]:
```
"""
This is my first script.
It prints the phrase "Hello, world."
"""
phrase="Hello, world."
print(phrase)
```

```
Hello, world.
```