COMP1811
Paradigms of Programming

Python Basics

Variables, Data Types and Operators

Dr. Yasmine Arafa

bit.ly/COMP1811_Python_Basics

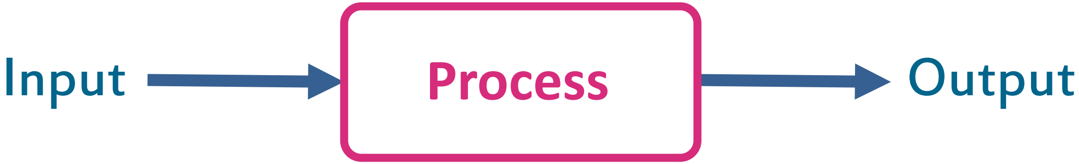
Adapted from https://www.cleanpng.com

On the Menu…

- Variables:
  - purpose and definition
  - memory allocation
  - assignment and reassignment
- Data Types
  - available data types
  - type casting
- Operators
  - arithmetic
  - comparison
- Python code structure fundamentals

Computer Programs
a general pattern

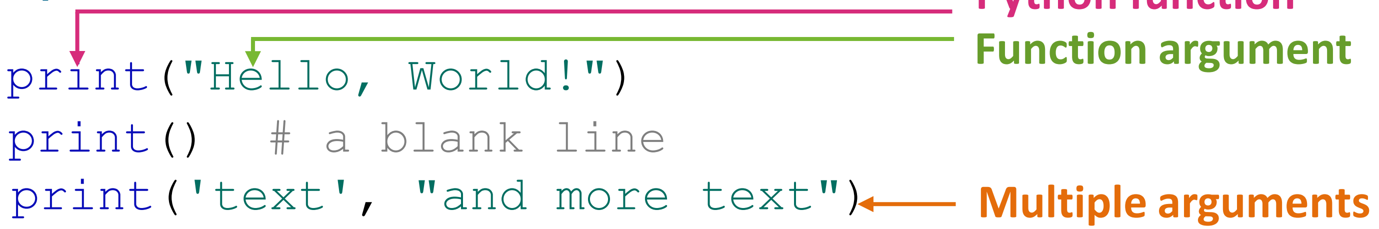
- General concept/pattern for most computer programs/applications:
  - take some input from the user or from another computer function;
  - process that input; then
  - output or display something back to the user (or the computer function).



- Simple mechanisms for input/output in Python are:
  - `input()`
  - `print()`

First Commands – Python I/O Commands
print()

- Displays output to the console



```
print("Hello, World!")
print()   # a blank line
print('text', "and more text")
```

Python function

Function argument

Multiple arguments

- Strings are any text between double or single quotes
- To embed quotes in the text, use the escape sequences \ before each quote

```
print("Suppose two swallows \"flock\" together.")
print('African or "European" swallows?')
print("""For text on multiple lines,
         use triple quotes""")
```

First Commands - Python I/O Command
input()

- Provides a way of getting keyboard input into the program.

```
print("Hello", input("Please enter your name:"))
```

  - will output the word "Hello" as well as what the user typed in:

```
Hello Alan
```

  - NOTE: input() returns text (a string value).

- Event-driven programming is another way of getting input from the user or other devices (mouse, touch screen, games, etc.).
  - more on this when we cover GUIs

There’s a Problem!

- All the following code does is display the entered name on the screen:

```
print("Hello", input("Please enter your name:"))
```

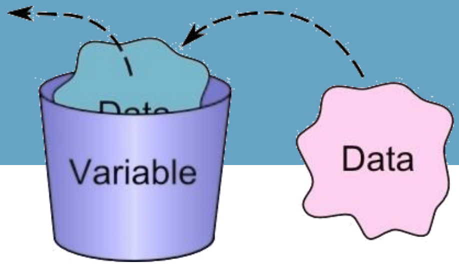
- What if the program needs to so something with the name?
  - i.e. process input,
  - for example: add the name to a list of existing names.
- So, the ***name*** **entered needs to be stored somewhere** so that the program can use it when it needs to…

How?

- To use the *name* entered elsewhere in the code, the program must store (temporarily) a copy of what the user has typed in.
- This is achieved with something called a **variable**:

```
name = input("Please enter your name:")
```

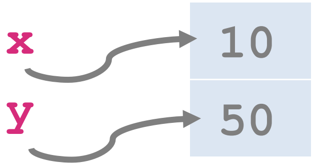
Variables



- A variable is:
  - a container of data (a value) that can change while the program runs;
  - this container is a place in computer memory which has been especially allocated to the variable (allocation is temporary, only while a program runs);
  - the place in memory is allocated when the variable is created/defined.
- A variable name (identifier) is:
  - a reference to the memory location allocated.
- Examples:

```
x = 10
y = 50
```

reference memory



x 10 10 is value in memory

y 50 50 is value in memory

Variable Assignment

- Syntax:

```
<variable_name> = <any value>
```

Angular brackets mean, replace what's between them.

- By convention, variable names begin with a lowercase letter.
  - examples: `name`, `module_code`, `result`

- To assign data to a variable, use the "=" sign
  - the data can be something the user has typed in **or** can be something fixed:

```
name = input("Please enter your name:")
module_code = "COMP1811"
```

name is entered by user

module_code is fixed in the code

  - the above is known as **assigning** a value to the variable.

Variable Assignment

cntd.

- **Note**: the "=" sign does not work the same as in math;
  - **in math**, read it as *is equal to*, for example: `PI = 3.14`,
  - **in programming**, read it as *is assigned the value*, example: `module_code = "COMP1811"`

- Multiple assignments on one line is **legal** in Python.
  - Example:

```
x = 10
y = 5
z = 20

x = y = z
```

✓ ← the value of x is assigned the value of the last variable assigned, i.e. the value of **x is 20 and y is 20**

Variable Assignment
cntd.

- More examples of multiple assignment:

```
x, y = 8, 10
print("x is ",x,"and y is ",y) #output: x is 8 and y is 10

x, y = y, x
print("x is ",x,"and y is ",y) #output: x is 10 and y is 8
```

Variable Re-assignment

- What is assigned to a variable can be changed by assigning it again (variables are mutable).

```
module_code = input("What is the module code?")
module_code = "COMP1811"
```

← assignment

← reassignment

- Re-assignment can change the variable type

```
module_code = 1811
```

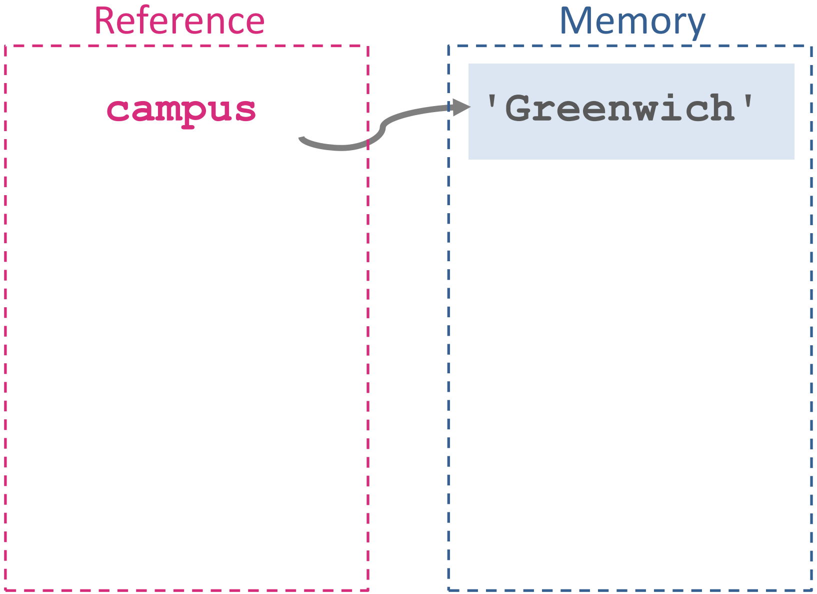
← now a number

  – this could cause problems in operations where types don’t match!

More on Variable Reassignment

examples

```
campus = 'Greenwich'
```



Reference

campus

Memory

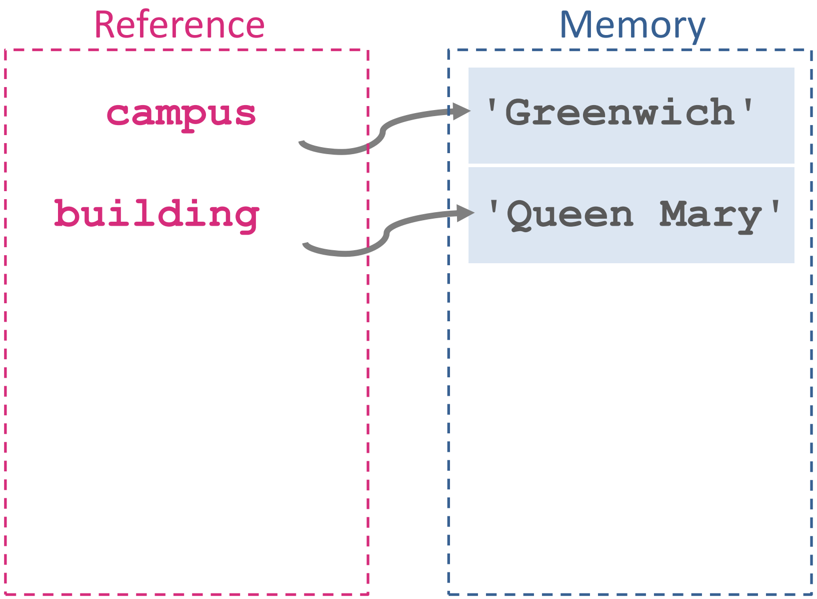
'Greenwich'

More on Variable Reassignment

examples

```
campus = 'Greenwich'

building = 'Queen Mary'
```



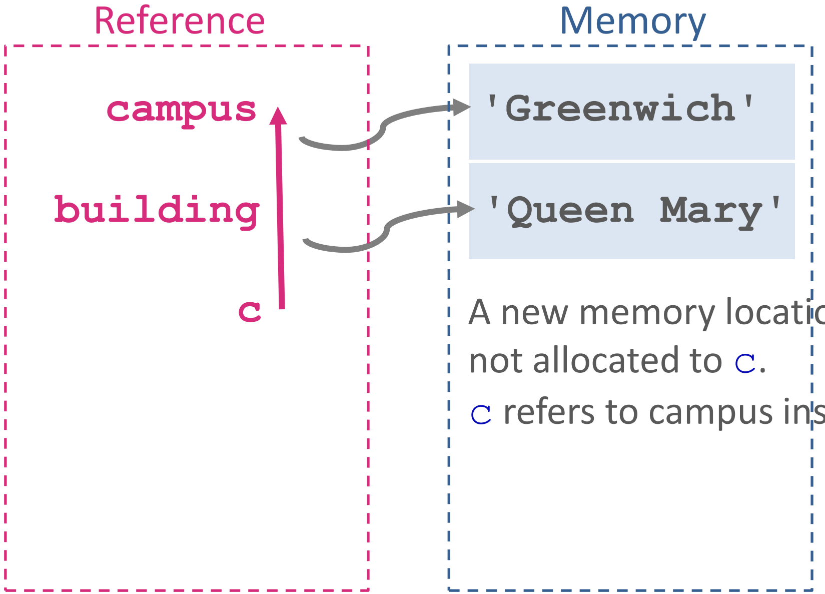
More on Variable Reassignment

examples

```
campus = 'Greenwich'

building = 'Queen Mary'

c = campus
```



A new memory location is not allocated to `c`.
`c` refers to campus instead.

More on Variable Reassignment

examples

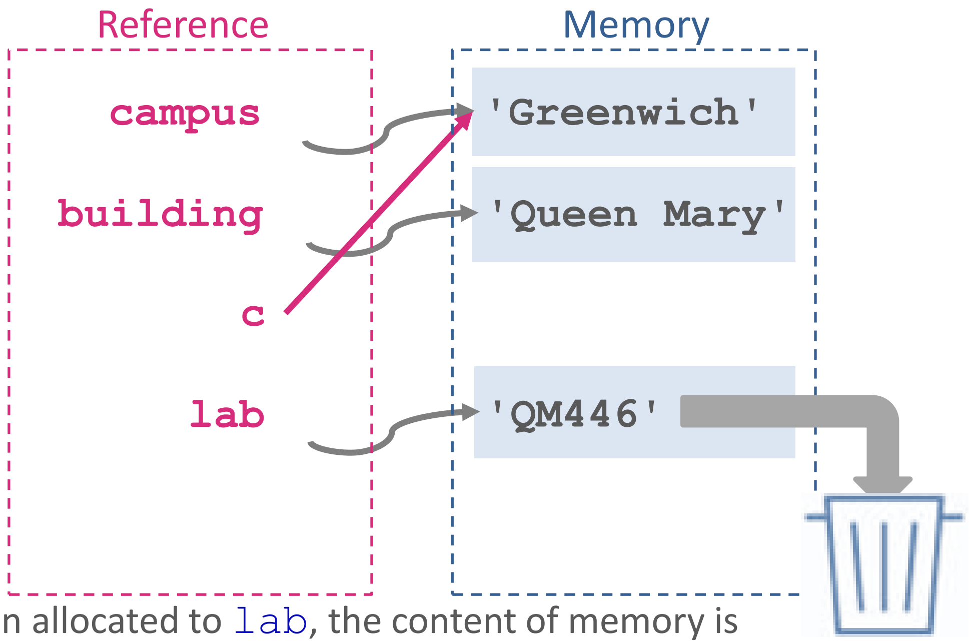
```
campus = 'Greenwich'

building = 'Queen Mary'

c = campus

lab = 'QM446'

lab = 105
```



Because a memory location has already been allocated to `lab`, the content of memory is deleted before the new value is stored in memory. Python collects the garbage and recycles the memory.

More on Variable Reassignment

examples

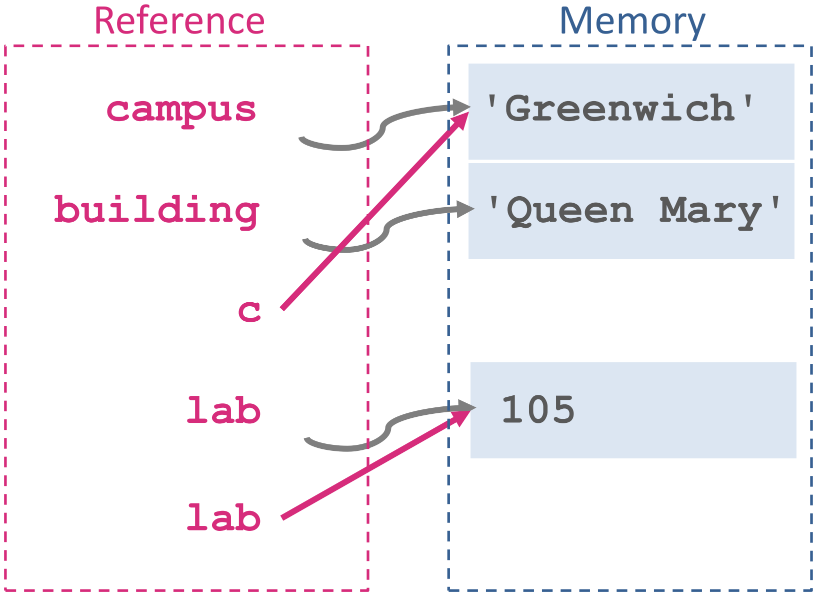
```
campus = 'Greenwich'

building = 'Queen Mary'

c = campus

lab = 'QM446'

lab = 105
```



Reference

campus

building

c

lab

lab

Memory

'Greenwich'

'Queen Mary'

105

Constants

- A constant is a type of variable that holds a value that does not change,
  - useful to store values you know will never change in the program,
  - examples:

```
PI = 3.14
MAX_GRADE = 100
HOST_CITY = "London"
```

- By convention, constants' names should be in uppercase letters.
- Unlike variables, constants should not change;
  - Python does not force this,
  - which means change should not occur *by convention* NOT because Python prevents the change, as is the case in other languages.

Python Variable Naming

some rules

- Python is case-sensitive:
  - e.g. `module_code`, `module_Code`, and `Module_Code` are three different names!
- Variable names can only contain alphanumeric characters and _:
  - must start with a letter,
  - names should be meaningful (e.g. `age` ✓, `a` ✗),
  - avoid names starting with _ (e.g. `_code` - it is legal but best to avoid✓),
  - white spaces are not allowed (`module code = 1811` ✗),
  - special characters (%, $, £, etc.) are not allowed (`%total = 58` ✗),
  - digits at start are not allowed (`10ml_price = 12` ✗).
- Variable names must not be a reserved or keyword word such as: `True`, `False`, etc. (`True = "Good result"` ✗)

Reserved Words (Keywords) in Python

- Keywords have predefined meaning and use in the language, and
  – cannot be used as identifiers for variables, functions, classes, methods, etc.
- Python keywords:

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

Common Variable Naming Conventions

PEP8

- Variable names (identifiers) should describe the purpose of the variable:
  - they should be meaningful within context, and
  - should avoid names like: `var1`, `button1`, etc.
  - names such as `save_btn`, or `exit_btn` are better names.
- Variable names should start with a lowercase letter:
  - beginning with an uppercase is legal BUT can be confused with a class name.
- With single word variable names, all characters are lowercase
  - e.g.: `grades, name`
- Multiple words (compound names) are separated with an underscore
  - e.g.: `module_code`, `savings_acct`, `current_acct`

Common Variable Naming Conventions
cntd.

- CamelCase name can be used BUT this is not strictly Python Style
  - camelCase ("dromedary case") is where Multiple words are separated by capitalising the first letter of each word except for the first word,
  - that is, the first letter of each word in a compound word is capitalised;
  - variables should have lowerCamelCase names (ie the first compound word is lowercase) e.g. `firstName = "James"`
- Constant names are typically all capital
  - e.g.: `MAX_SIZE, PI, VALUE_ADDED_TAX`
- Details of Python best practice and naming convention can be found at PEP8.

Data Types

Data Types

- A Data Type is:
  - A category characterising a single or set of data values, and
  - constrains the operations that can be performed on that data.

- Python data types:
  - **str** (strings → text)
  - **int** (integers → whole numbers)
  - **float**ing point numbers (decimals)
  - **boolean** (true/false values)
  - ***complex numbers*** *(not covered)*
  - user defined – UDT (covered later)

- Examples:

```
x = 5.5                # defines a float
y = 1                  # defines a integer
```

Python Typing
Where’s My Type?

- A variable takes a value that has a particular data type:
  - but you don’t need to declare that type before the variable can be used,
  - Python knows the type of a variable, even if you don't!

- Python is a strongly dynamically typed language.
  - Strong **typing**:
    means that variables do have a type and that the type matters when performing operations on a variable

Python Typing
Where’s My Type?, *Cntd.*

– Dynamic Typing:

Means the type of the variable is only determined at runtime:

- No need to declare a variable/constant or give it a type before use.
- Variable/constant data types are determined based on their value’s type.
- All type checking is done at runtime.

```
x = 5.5                    # defines a float type
```

Python Typing

- Example variable declarations:

```
name = "Alan Smith"  # python binds a string type
flag = True          # boolean
x = 10.5             # float
y = 6                # integer
```

- To find out a variable’s data type, use the Python function `type()`:
  - Examples:

```
type(1)       # returns integer (int)
type("Hello") # returns string  (str)
type(x)       # returns float   (float)
```

Python Typing

Cntd.

- Variable/constant types can be overridden at any time!
  - Examples:
    ```
    x = 4
    x = "four"
    x = 55.8
    ```
- Python is strongly typed
  - Obviously, Python isn't performing **static** type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
  - Example: `4 + "four"` ✗ not going to work!
- Let's start by looking at Python's built-in data types next.

Strings

- Built-in sequence type – a string is a list of characters (text)

```
forename = "Jon"
surname  = 'Alan'
```

- Things you can do with strings
  - Concatenation – joins 2 strings together:

```
full_name = forename + surname
```

  - Getting the length of a string:

```
len(full_name)
#returns the number of characters in string
```

Output

```
JonAlan
```

```
7
```

Numeric Types

- Built-in types that represent numeric values.
- The subtypes are `int`, `float` and `complex`.
  - `long()` available in Python 2 – deprecated in Python 3
- All numeric types support the typical arithmetic operations you'd expect to find to perform calculations.
- Mixed numeric type operation is supported, with the "narrower" type widened to that of the other.
  - i.e. an integer is widened to a float

```
x, y = 10.5, 3
z = x + y               # results in a float
```

Boolean Types

- Built-in data type that represents one of the two values `True` or `False`.
- Generally, used to represent the value resulting from comparison operations and conditionals.
- Examples:

```
result = 12 < 5
print(result)
```

Output

```
False
```

Converting Data Types

casting

- Casting converts one data type to another
  - that is done by using the constructor for the data type you want to convert to

- Examples:

– int to float:	`float(9)`	→	`returns 9.0`
– float to int:	`int(5.3)`	→	`5`
– str to float:	`float("10")`	→	`10.0`
– int to str:	`str(10)`	→	`"10"`
– float to str:	`str(10.10)`	→	`"10.1"`
– str to float:	`float("10")`	→	`10.0`
– str to int:	`int("six")`	→	error

Python Data Types and Memory Allocation

- Data types determine the memory size that is allocated to a variable when it is declared:

Type	Description	Size
`int`	Integer (whole number)	24 bytes
`float`	Real number (decimal)	24 bytes
`boolean`	logical values (True/False)	24 bytes
`string`	Set of characters	1 byte per character + 37 overhead

- All data types in Python are objects
  - which means they have an associated set of methods (functions) that can be used to manipulate them (e.g. `len()` for strings).

Operators

Python Operators

- Operators are special symbols in Python that carry out
  - arithmetic (e.g. +, -, /, etc.)
  - or logical computation.
- The value that the operator operates on is called the *operand*:
  - example: `x + y`
- A sequence of operands and operators is called an *expression*:
  - example: `x * 10 + y`
- A line of code that assigns an expression to a variable is called a *sentence*:
  - example: `result = x * 10 + y`

Arithmetic Operators

- Used to perform mathematical operations on variables and values:

Operator	Description	Example	Value of num	
`=`	Assignment	`num = 7`	**7**	
`+`	Addition	`num = 2 + 2`	**4**	
`-`	Subtraction	`num = 6 - 4`	**2**	
`*`	Multiplication	`num = 5 * 4`	**20**	
`/`	Division (true division)	`num = 9 / 2`	**4.5**	
`%`	Modulo (modulus)	`num = 9 % 2`	**1**	
`//`	Floor division	`num = 9 // 2`	**4**	
`**`	Exponentiation	`num = 9 ** 2`	**81**	
		`num = 4 ** 0.5`	**2.0**	(square root)

Arithmetic Operators
modulus vs floor division

- Modulus: is the remainder after dividing one number by another
  - example:

```
11 % 2 → 11 / 2  → 5 and remainder 1 → 1
40 % 4 → 40 / 4  → 10 and remainder 0 → 0
```

  - useful for determining whether a number is odd or even.

- Floor division: yields the quotient, not the remainder.
  - example:

```
11 // 2 → 11 / 2  → 5 and remainder 1 → 5
40 // 4 → 40 / 4  → 10 and remainder 0 → 10
```

  - useful for when you need to the result of division as a whole number.

Order Matters!

order of precedence

- With compound expressions such as `n + 2 * x ** 4` the mathematical order of calculation applies:
  - **First level of precedence:** top to bottom in precedence table
  - **Second level of precedence:** if there are multiple operations that are on the same level then precedence goes from left to right.

- Example:

```
x = 3 * 2 ** 3
```

vs

```
x = (3 * 2) ** 3
```

Precedence Table	
()	Brackets (inner before outer)
**	Exponentiation
*, /, //, %	Multiplication, division, floor, modulo
+, -	Addition, subtraction
=	Assignment

Arithmetic Expressions

- Examples:

```
x = 3 * 2 / 3

y = x ** 4

z = ((x * y) // 2) + (y + 1) - x / y


z = z + (y * 5)

z += y * 5
```

← both lines are equivalent,
but last line uses syntactic sugar!

Short-cut Operations

Syntactic Sugar

- Syntactic Sugar, in programming, is a way to make things easier by reducing the amount of code:

```
something += 10
something -= 10
something *= 10
something /= 10
something **= 10
something %= 10
```

Equivalent to ⟺

```
something = something + 10
something = something - 10
something = something * 10
something = something / 10
something = something ** 10
something = something % 10
```

Operations on Strings

string multiplication

- Python strings can be multiplied by an integer:
  - the result is many copies of the string concatenated together;
  - the **result is always a string**.

- Examples:

```
a_string = "Hi!"
result = a_string * 3
print(result)

a_string = "ha "
print(6 * a_string)

print(2 * 4 * "8")
```

Output

```
Hi!Hi!Hi!
```

```
ha ha ha ha ha ha
```

```
88888888
```

Operations on Strings

string concatenation

- Concatenation adds strings together:
  - **add** here means "join" strings together
  - the **+** operators is used for concatenation,
  - example:

```
a_string = "I will finish my coursework by "
date = "22/01/24"
print(a_string + date + ".")
```

Output

```
I will finish my
coursework by 22/01/24
```

Operations on Strings
string concatenation, *Cntd.*

- Strings cannot be concatenated with and integers :
  - example:

```
print("5" + 2)
```

Output

```
runtime error
```

  - workaround:
    - convert the number to a string first, then concatenate it:
    - example:

```
num = str(2)          # converts a value into a string
print("5" + num)      # prints both values on the same line
```

"52"

Comparisons Operators

- Are relational operators which compare the values of operands and return a boolean result:

Description	Expression	Result
Less than (<)	`2 < 4`	**True**
Greater than (>)	`x > y`	**False** (where `x=1, y=10`)
Equal to (==)	`2 == 4`	**False**
Not equal to (!=)	`2 != 4`	**True**
Less than or equal (<=)	`2 <= 4`	**True**
Greater than or equal (>=)	`4 >= 4`	**True**
Multiple comparisons	`2 < 4 > 1`	**True**

- Remember: single "=" is assignment and double "==" is comparison.

Comparisons Operators

working with strings

- Python compares character by character based on their unicode value:
  - each character has a unique code that represent it as a hex value,
  - the unicode for lowercase letter is greater than the unicode for uppercase.
- String comparisons are case sensitive

Description	Expression	Result
Less than (<)	`"two" < "three"`	**False**
Greater than (>)	`"two" > "three"`	**True**
Equal to (==)	`"two" == "Two"`	**False**
Not equal to (! =)	`"two" != "three"`	**True**
Greater than or equal (>=)	`"two" >= "three"`	**True**
Multiple comparisons	`"two" < "three" > "six"`	**False**
Unicode for "a" is greater than "A"!	`"a" < "A"`	**False**

Python Program Basics

Some Python Program Fundamentals
whitespace

- Whitespace is significant in Python.
  - where other languages may use {} or (),
  - Python uses indentation to denote code blocks.
    - *more on block next week…*

  - Adding whitespaces to the body of your code as in the example above will cause run-time error.

```
print("Hello World!")
    print("AND a special hello to COMP1811...")
```

causes run-time error!

Some Python Program Fundamentals

comments

- Comments are text notes about the code and are used for
  - for providing explanatory notes about the code:
    - this helps other programmers understand your code and its purpose, and
    - also helps you remember your own code if you’ve left it for a while;
  - for keeping versions of parts of the code that are not working properly but you want to keep.
- A comment is any statement in Python that begins with a #
  - the Python interpreter ignores comments (any text beginning with #)
- Single-line comments denoted by #
  - lines beginning with #, comment the entire line, and
  - partial lines beginning with #, comment the line from the # to the end of line.

Some Python Program Fundamentals
comments

- It is good practice to include in all your program files:
  - a **header comment** (such as in the previous slide) that includes details of who and when the code was created, its version and a brief description of its purpose; and
  - **in-line comments** that describe line or blocks of code.
- Comments should express information that cannot be expressed in the code – do not restate code.

Multi-line Comments

- Multi-line comments can be a collection of separate lines, each starting with #:

```
##
# Author: Some Programmer
# Date created: 02/10/23
# Date updated: no updates
# Version: 1.0
# Description: Demonstrates
#      the use of comments.
##
```

Multi-line Comments
cntd.

- Alternatively, define multi-line comments by enclosing them between three quotes at the beginning and end of the lines:

```
"""
  Multi-line documentation
  more text
  and more text.
"""
```

  - this technique doesn't create *true* comments - it simply inserts a text constant that does nothing,
  - be careful where you place them in the code - they could turn into **docstrings** and what was intended as a simple comment will become associated with an object and take up memory. They are better avoided.

In the lab today …

- There is a list of tasks to complete.
  1. Revise the Python Basics notebook and repeat the activities until you're competent in coding them.
  2. Take the quiz on Python Basics.
  3. Complete the Python exercises in Lab sheet 1.01. You will need to download and unzip the code needed for these exercises.

Next week…

- Boolean Logic
- Boolean Expressions
- Flow control / branching
  - Conditional statements
    - *if else*
    - *if elif else*
    - *nested if statements*

Further Reading

- **w3schools pages** - we have not covered everything there but this will give you more insight into how variables work.
  - Python variables
  - Casting data types
  - Python numbers
  - Python operators
- Online Python Tutorials
  - LearnPython.org — Interactive tutorials providing a comprehensive introduction to coding in Python. Suitable for complete beginners.
  - Python at TutorialPoint — Another good tutorial starting from Python basics and leading onto more advanced topics.