CMPUT 274

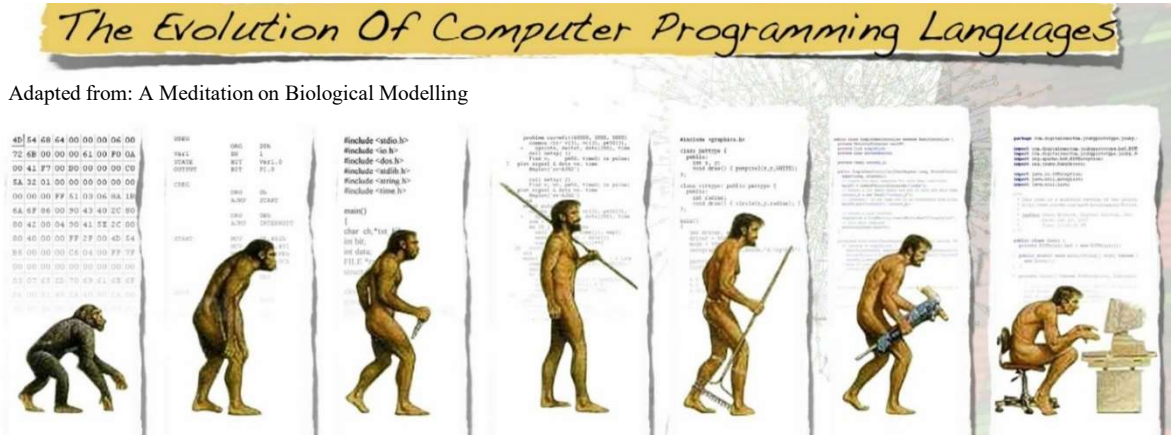
Python Basics

Topics Covered:

- Interpreted vs compiled code
- Programming style: comments, PEP8
- Simple input/output
- Values and variables
- Introduction to built-in data types and how to use them

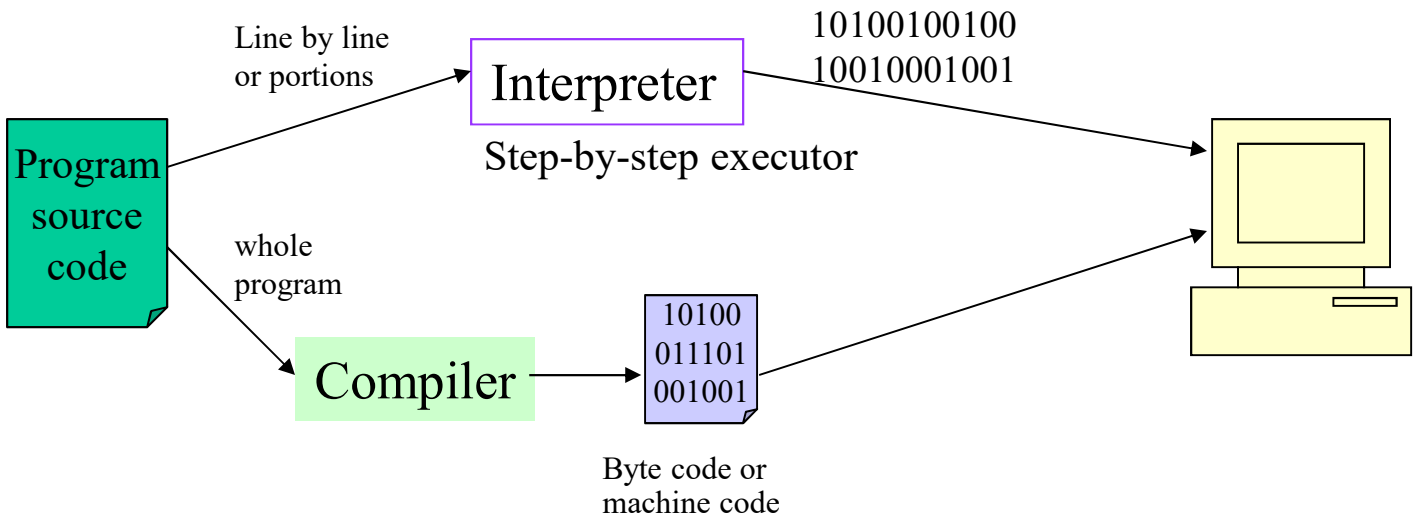
Programming

- program = set of instructions given to a computer
- Computer understands machine language (binary number codes: 1s and 0s) specific to its CPU
- Higher level programming languages were developed to make programming easier
  - "Human readable" code
  - Processor independent



Translate Source Code

- But the CPU only understands machine language, not higher level languages!
- Need either an interpreter or compiler to translate high-level instructions into machine language



Interpreted vs Compiled

- Python is interpreted
  - Start interpreter by typing python3 in interactive shell
  - Translates program line-by-line until meets its first error or the end of program
  - Code is interpreted EVERY time you run your program

- C++ is compiled
  - Translates entire program into machine code efficiently → execution time is generally faster
  - Code is only compiled when new executable is required → e.g. change to code, different kind of machine

Python Interpreter in Action

- start Virtual Machine (VM)
- open new terminal
- launch interpreter: type python3 (or ipython3)
- display Hello World on screen using print **function**

  `help(print)` ← use help function to look up documentation

  press q to return to interpreter

  `print("Hello World!")`

Can use double quotes or single quotes to enclose Python strings. **Just be sure they match!**

- quit interpreter

  `exit()` ← one way of many to quit interpreter

First Python Program

- open editor from terminal

  `subl helloWorld.py` ← starts Sublime Text Editor

```
# ================================
# Name: Megan Flanders
# ID: 123456
# Collaborated with: Zachary Friggstad
# CMPUT 274, Fall 2019
#
# Practice 1: First program
# ================================

print('Hello World!')
```

Single line comment.
Ignored by interpreter.

- interpret and run python program from terminal

  `python3 helloWorld.py`

Style in Python

- PEP8: Style Guide for Python Code
  - Covers formatting, comments, naming conventions
  - Invoke style checker from terminal: `style helloWorld.py`

- Comments
  - Improve code readability and maintainability
  - Should explain approach of code (the why)
    → not just a line-by-line description
  - single line: start with #
  - multi-line: used to automatically create code documentation

```
"""
comment over
many lines
"""
```

enclosed by three quotes

More Simple Output

- start Python interpreter and try:

```
>>> print("CMPUT","274")
CMPUT 274
>>> print("CMPUT","274", sep="-")
CMPUT-274
>>> print("CMPUT","274", end="-")
CMPUT 274->>>
This will print text
on 2 lines.
```

- Escape sequences: examples

\n	newline
\t	tab
\\	backslash (\ )
\"	double quote (")

Prompt User for Input

- `input("custom prompt")`
  - Displays *custom prompt* on screen
  - Waits for user to enter value (always read in as a string)
  - End of the string is indicated when user presses Enter key

- Try it yourself:

```
>>> input('Enter number: ')
Enter number: 34
```

- But how does the program use the value entered by the user?

- Save the value by assigning it to a variable.

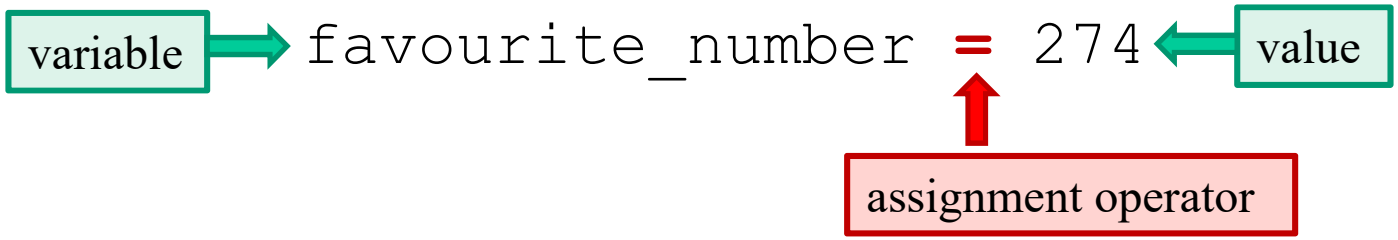
Values and Variables

- Values are stored in main memory
- Each value has a type associated with it:
  - Integer: `274`
  - Float: `3.1415926535897931`
  - Boolean: `True`
  - String: `"This is a string."`
  - List: `["CMPUT", 274, -91.0, 'A', False]`
  - etc.

  → The type is stored next to the value in memory
- Can use `type()` to check a value's type
- Variables are convenient way to access and/or manipulate values in memory

Values and Variables

- To assign a value to a variable, use assignment operator:



- Python is dynamically typed
  - → do not have to explicitly declare a variable along with its type (this is different from C++)
  - → **CAUTION:** when updating a variable, it is possible to change the type of value that variable is referring to

- Any value not associated with a variable is periodically deleted from memory by Python's garbage collector

Variable Names

- Python variable names can contain letters, numbers, and underscores ('_')

  → generally should start with letter

- Python variable names are case sensitive

  → `myVariable` is different from `MyVariable`

- Python **keywords** cannot be used as variable names:

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

Variable Names

- Variable names (identifiers) can be arbitrarily long
- In general: choose concise, but descriptive identifiers

  → self-document code

```
a = b * c
```
vs.
```
increase = salary * percent
```

- Single character identifiers are typically used for iterating
- Can use multi-word identifiers. Many conventions:
  - Underscore

```
weekly_pay = hours_worked * pay_rate
```

  - Lower camel case

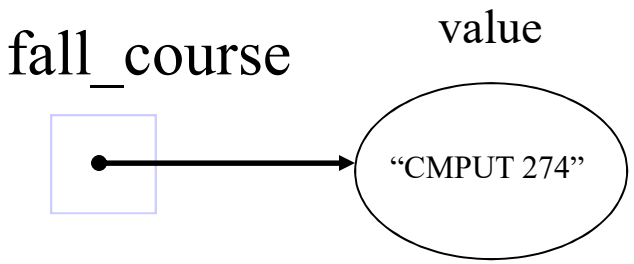
```
weeklyPay = hoursWorked * payRate
```

  - Upper camel case

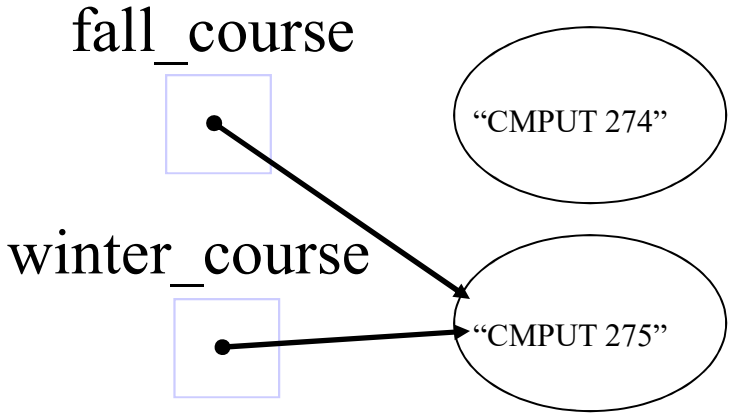
```
WeeklyPay = HoursWorked * PayRate
```

Variables are References

```
fall_course = "CMPUT 274"
```




```
fall_course = "CMPUT 275"
winter_course = "CMPUT 275"
```



What happens to the first value "CMPUT 274"?

Built-In Type: `int, float, complex`

- **Integer and Float** are numbers [immutable]
  - Operators +, -, *, /, //, %, and ** perform addition, subtraction, multiplication, division, floor division, modulo, and exponentiation respectively
  - Examples:
    - 10+23 *33*
    - 67-5 *62*
    - 20+3*60 *200*
    - 210/60 *3.5*
    - 7//3 *2*
    - -7.0//3 *-3.0*
    - 7%3 *1*
    - 2**10 *1024*

Example functions:
abs(x) → absolute value of x
int(x) → converts x into integer
float(x) → converts x into floating point
str(x) → converts x into a string

Convert Type

- Can convert from one type to another by using built-in conversion functions to create new object:
  - `int()`, `float()`, `str()`, `list()`, etc.

- Recall that `input()` reads in a user's entry as a string:

```
>>> a=input('Enter number: ')
Enter number: 34
>>> b=a*2; print(b)
3434
>>> a=float(a); b=a*2; print(b)
68.0
```

- Can't mix types when performing an operation:

```
>>> num = 274
>>> print("CMPUT " + str(num))
CMPUT 274
```

String concatenation

Built-In Type: `bool`

- **Boolean** can be either True or False [immutable]
  - Operators and, or, not for conjunction, disjunction and negation

x	not x
False	True
True	False

p	q	p and q	p or q
False	False	False	**False**
False	True	False	True
True	False	False	True
True	True	**True**	True

  - Comparison operators:
    - == equality
    - != not equal
    - < less than
    - > greater than
    - <= less than or equal
    - >= greater than or equal

```
>>> False or True
True
>>> not (False or True)
False
>>> 7==2013
False
>>> 7!=2013
True
>>> (2013>=7) and (2013<=2020)
True
```

Built-In Type: `str`

- **String** is a sequence of characters [immutable]
  - Values of strings are written using quotations
    - "CMPUT" or 'CMPUT'
  - Characters are indexed starting from 0.
    - my_var="CMPUT"; my_var[2] returns 'P'
  - Operator + performs concatenation
    - "ABC"+"CDE" results in "ABCCDE"
  - Sample string **methods**:
    - count(item)
    - find(item)
    - split(char)
    - upper()
    - lower()
    - center(w,char)
    - strip()
    - replace(old,new,max)
    - isdigit()
    - islower()
    - etc…

```
>>> my_var="CMPUT"
>>> my_var.lower()
'cmput'
>>> my_var.find('P')
2
>>> my_var.split('P')
['CM', 'UT']
>>> " CMPUT  ".strip()
'CMPUT'
```

String Method `format()`

- Allows insertion of value into a string at placeholder {}
- Allows formatting of that value within the string
- e.g. `"The rental costs ${0:5.2f}".format(price)`

- **Example Format Modifiers**
  - Number {:15} field width (right-justified)
  - < {:<15} left-justified in field width
  - 0 {:015} pad with zeros
  - . {:15.2f} digits after decimal point

```
>>> a=23; b=100
>>> "My numbers are {1} and {0}".format(a,b)
'My numbers are 100 and 23'
>>> name = 'Fred'; amount = 5.43
>>> print('The person {0:^015} has {1:>07.2f} dollars'.format(name, amount))
The person 00000Fred000000 has 0005.43 dollars
```

Built-In Type: `list`

- **List** is a sequence of values of *any* type [mutable]
  - Elements in a list are numbered starting from 0
    - Access an element of a list via its index **k[index]**
  - Operators + and * concatenate and repeat sequences respectively
    - **[1,2,3] + [4,5,6]** results into [1,2,3,4,5,6]
    - **[1,2,3] * 3** results into [1,2,3,1,2,3,1,2,3]
  - Operator : slices in a list
    - **k=[1,2,3,4,5,6]; k[2:4]** results in [3,4]
    - **k=[1,2,3,4,5,6]; k[2:]** results in [3,4,5,6]
    - **k=[1,2,3,4,5,6]; k[:4]** results in [1,2,3,4]
  - Membership operator in asks whether an item is in a list
    - **3 in [1,2,3,4,5,6]** returns True
  - Length of a list with function len
    - **len([1,2,3,4,5,6])** returns 6

Sample List Methods

- append(item) adds an item at end of list
- insert(i,item) inserts item at i^{th} position of list
- extend(iterable) appends all the items in the iterable
- pop() removes and returns last item in list
- pop(i) removes and returns i^{th} element in list
- del(i) removes i^{th} element in list
- remove(item) removes 1^{st} occurrence of item
- sort() modifies list to be sorted
- reverse() modifies list to be in reverse order
- count(item) returns the number of occurrences of item in list
- index(item) returns the index of 1^{st} occurrence of item

Lists and Strings

- list

```
>>> list("CMPUT")
['C', 'M', 'P', 'U', 'T']
```

- split

```
>>> "1,2,3,,5".split(',')
['1', '2', '3', '', '5']
>>> "the cat sat on the mat".split()
['the', 'cat', 'sat', 'on', 'the', 'mat']
>>> "the,cat,sat,on,the,mat".split(',',3)
['the', 'cat', 'sat', 'on,the,mat']
```

- join

```
>>> ' '.join(['1', '2', '3', '4', '5'])
'1 2 3 4 5'
>>> ''.join(['1', '2', '3', '4', '5'])
'12345'
>>> '**'.join(['1', '2', '3', '4', '5'])
'1**2**3**4**5'
```

Built-In Types: `tuple, set`

- **List** is a mutable heterogeneous sequence of values
  [2, True, "cat", [1,2,3], 3.5, 2]

- A **tuple** is an immutable list

  (2, True, "cat", [1,2,3], 3.5, 2)

  → Like for strings, you would get an error if you try to change the content of a tuple. *BUT you can change mutable objects inside of tuple; e.g. the list inside of the above tuple*

- A **set** is an unordered collection of unique immutable objects, but the set itself is mutable {2, True, "cat", 3.5}

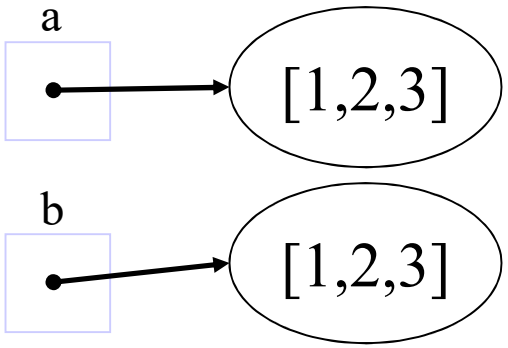
  → A set does not support indexing (is not sequential)

  → Sets support methods such as union (|), intersection (&), issubset (<=) and difference (-), as well as add(item), remove(item), clear() and pop().

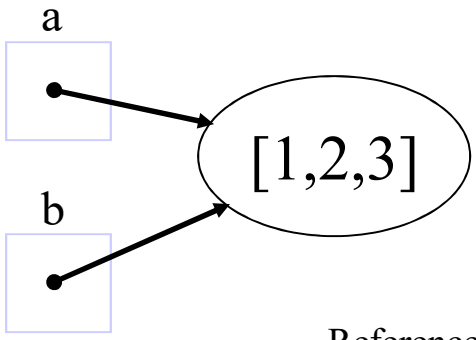
Aliasing

`x = y` does not make a copy of y
`x = y` makes x reference the same object y references

```
>>>a=[1,2,3]
>>>b=[1,2,3]
>>>a is b
False
```




```
>>>a=[1,2,3]
>>>b=a
>>>a is b
True
```

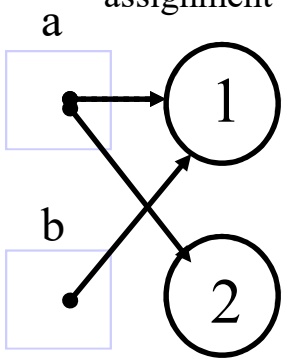


Beware:

```
>>> a=[1,2,3]; b=a
>>> a.append(4)
>>> print(b)
[1,2,3,4]
```

```
>>> a=1; b=a
>>> print(b)
1
>>> a=a+1; print(b)
1
```

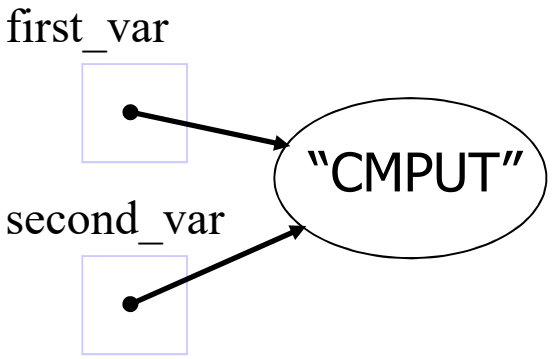
Reference lost after assignment



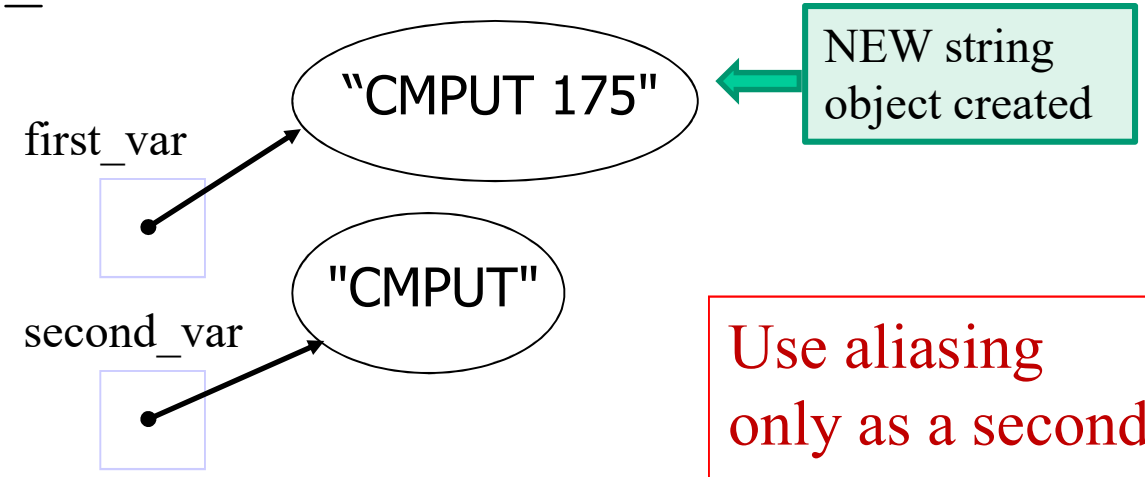
New object created

Aliasing Can Cause Problems

```
first_var = "CMPUT"
second_var = first_var
```




```
first_var = first_var +" 175"
```



Use aliasing only as a second name for a **mutable** object.