1811/2807/7001ICT
Programming Principles

School of Information and Communication Technology
Griffith University

Trimester 1, 2024

6 Variables

In this section we learn how to store values with variables.

We also peek under the hood at the object-oriented nature of Python, and how variables are references and their values are objects.

6.1 Using variables to store values

In the REPL or the interpreter, assignment statements allow us to use variables to store values for later use, perhaps multiple times.

For example, calculating molecular weights in the REPL, we'll start by saving the atomic weights of Hydrogen and Oxygen.

```
$ python3
>>> H = 1.00794
>>> O = 15.9994
>>>
```

`H` and `O` are new names we just made up.

= is the assignment operator. It binds the name on its left to the value on its right. The value on its right is evaluated before the binding happens.

Now we can recall the values by either just typing the name or printing it.

```
>>> H
1.00794
>>> print(O)
15.9994
>>>
```

Now we can calculate the molecular weights of water (H_2O) and hydrogen peroxide (H_2O_2).

```
>>> H * 2 + O
18.01528
>>> 2 * (H + O)
34.01468
>>>
```

6.2 Names for variables

Python names (identifiers) must be created following some rules:

- They must start with a letter, or underscore (`_`).
- They may continue with more letter, digits, or underscores.
- They are case sensitive. `H` and `h` are different names.
- They must not be one of Python's reserved words:

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

There are some conventions that we must follow too:

- Don't start names with underscores until we start programming with classes and you know what it means.
- If you only intend to give a variable one value and never change it, that is keep it constant, give it a name in all capitals.

  Examples: `H`; `O`; `H2O`; `SIZE`; `SCREEN_WIDTH`.

  Use underscores to separate the words in long constant names.
- If you intend a variable to take different values as your program runs, make it start at least with a lower case letter. Examples: `x`; `count`; `mi2km`; `currentMouseLocation`.

  Use snake case or camel case to separate the words in long variable names.

6.3 Object-orientation

Programming in Python is easier if you appreciate its object-orientation early.

> *Definition*: A class is a data type that defines the properties for any objects that are instances of this class.

> *Definition*: An object is a specific instance of class. There may be many objects that are instances of a particular class. An object stores values in memory.

> *Definition*: A reference is a value that identifies a particular object.

6.3.1 Values are objects and their types are classes

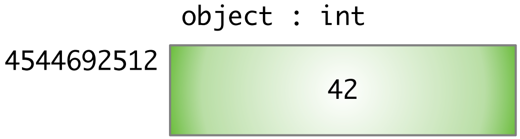
The values in Python, as we use the REPL or as we run a script with the interpreter, are saved in memory (RAM) as objects.

We know that all values have a type, and that the type is a class.

We can even find out where in memory the value/object is with the `id` function.

```
>>> 42
42
>>> type(42)
<class 'int'>
>>> id(42)
4544692512
>>>
```

To illustrate this:



An object with type `int`, stored in memory at locations starting at memory location `4544692512`, contains the value `42`.

Or put another way, the object at location `4544692512` is an instance of class `int` and contains the value `42`.

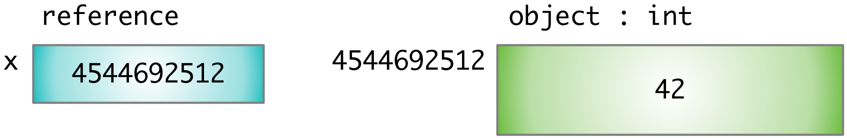
6.3.2 Variables are names for references

```
>>> x = 42
>>> x
42
>>> type(x)
<class 'int'>
>>> id(x)
4544692512
>>>
```

First we assigned (bound) the value `42` to the variable `x` with an assignment statement.

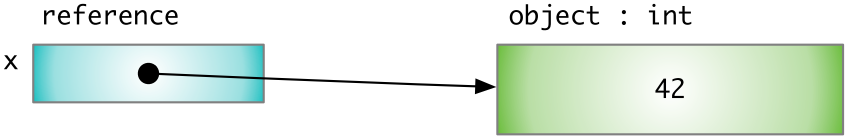
Then when we use the variable name `x`, it has all the same properties as the value `42`.

To illustrate this:



The variable name `x` labels a reference which contains the memory location of the instance of class `int` that contains the value `42`.

The actual memory location will be different every time we try this experiment, so lets drop it from the diagram and use an arrow to indicate the relationship instead.



6.4 Using variables

Lets start the REPL afresh, and see what the variable `x` contains.

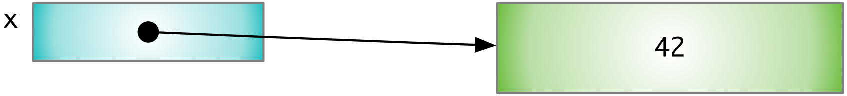
```
$ python3
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

It is an error to try to use a variable name that hasn't been assigned anything yet, because it does not exist!

Let's create it.

```
>>> x = 42
>>> x
42
>>>
```

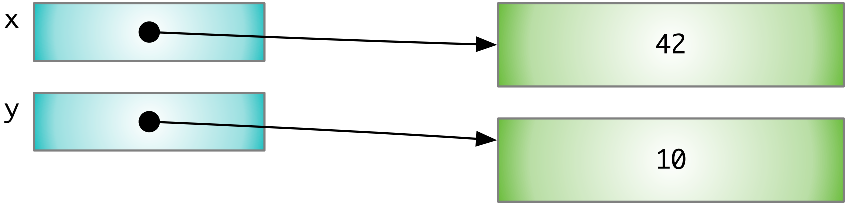
To save space, we'll drop the some of the labels from the diagram.



Let's create another variable `y`, assigning it a different value.

```
>>> y = 10
>>> id(x)
4510171424
>>> id(y)
4510170400
>>>
```

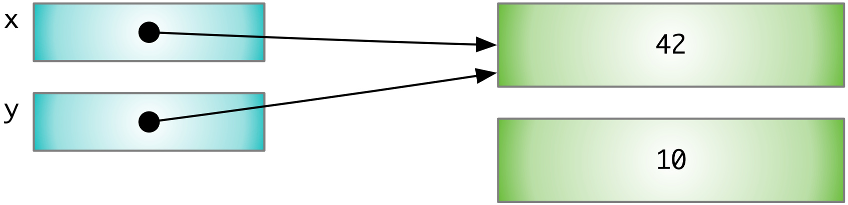
Since their `id`s are different, they are different objects.



Now let’s assign `x` to `y`, and see what happens.

```
>>> y = x
>>> id(x)
4510171424
>>> id(y)
4510171424
>>>
```

They are now referring to the same object, and the other object is *garbage*.



Definition: *Garbage* is a name given to objects and other data structures that are no longer needed.

If there are no references pointing to an object, the program can no longer use them.

The Python interpreter may safely remove them in a process called *garbage collection*.

Let’s add 1 to `x`, and see what happens.

```
>>> x = x + 1
>>> x
43
>>> y
42
>>>
```

The statement `x = x + 1` says take the current value of `x`, add `1` to it, and assign the result to `x`.

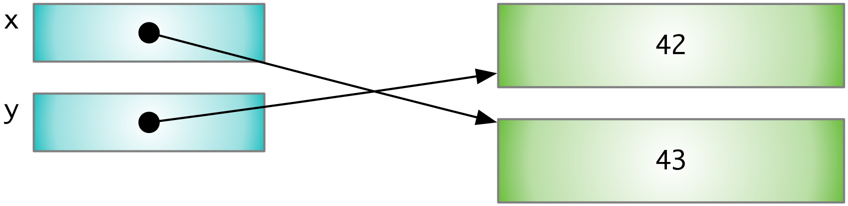
`x` now refers to an object with `43` in it.

`y` still refers to an object with `42` in it.

```
>>> id(x)
4510171456
>>> id(y)
4510171424
>>>
```

`x` and `y` now refer to different objects again.

Only `x`, the variable on the left of an =, changed.



6.4.1 Immutable objects

In the above examples, all of the variables referenced `int` objects.

When we changed the variables, the references changed, but *never* the objects containing the values.

This is a property of the `int` objects. The values they contain *never* change.

> Definition: To *mutate* is to change.
>
> If a thing can be changed, it is *mutable*.
>
> If a thing can not be changed it is *immutable*.

The types/classes we have met so far, `int` and `float`, are both immutable.

The simple types in Python are immutable, but we will meet more complex types that *are* mutable.

6.4.2 Variables don’t have types, objects do

In languages that are compiled, such as C, Java, Haskell, Swift, each variable has just one type.

In Python, like JavaScript, the variables are just names for references.

The types belong to the objects/values.

```
>>> x = 42
>>> x = 6.02e23
>>>
```

Compiled languages are usually *statically* typed at compile time.

Interpreted languages are often *dynamically* typed. The types of variables are bound at run time, and can change.

6.4.3 Deleting variables

Variables no longer needed may be forgotten, with `del`.

```
>>> a = 3
>>> a
3
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

Deleting the variable deletes the reference.

It will not affect other variables that reference the same object.

Section summary

This section introduced:

- variables as stores for values;
- using variable values;
- naming rules and conventions for variables;
- the `id` function, for identifying an object's identity by its location in memory;
- the object-oriented relationship beween variables (as references) to values (as objects, as instances of classes);
- how changing variables is more about changing references than values when objects are immutable; and
- deleting variables.