

PYTHON PROGRAMMING

Python

- Created in 1991 by Guido van Rossum (now at Google)
- Useful as a **scripting language**
 - **script**: A small program meant for one-time use
 - Targeted towards small to medium sized projects
- Used by:
 - Google, Yahoo!, Youtube
 - Many Linux distributions
 - Games and apps (e.g. Eve Online)



Installing Python

Windows:

- Download Python from <http://www.python.org>
- Install Python.
- Run **idle** from the Start Menu.

Mac OS X:

- Python is already installed.
- Open a terminal and run `python` or run Idle from Finder.

Linux:

- Chances are you already have Python installed. To check, run `python` from the terminal.
- If not, install from your distribution's package system.

Installing Python

- Python distribution is available for a wide variety of platforms.
- Download only the binary code applicable for your platform and install Python.

Unix and Linux Installation

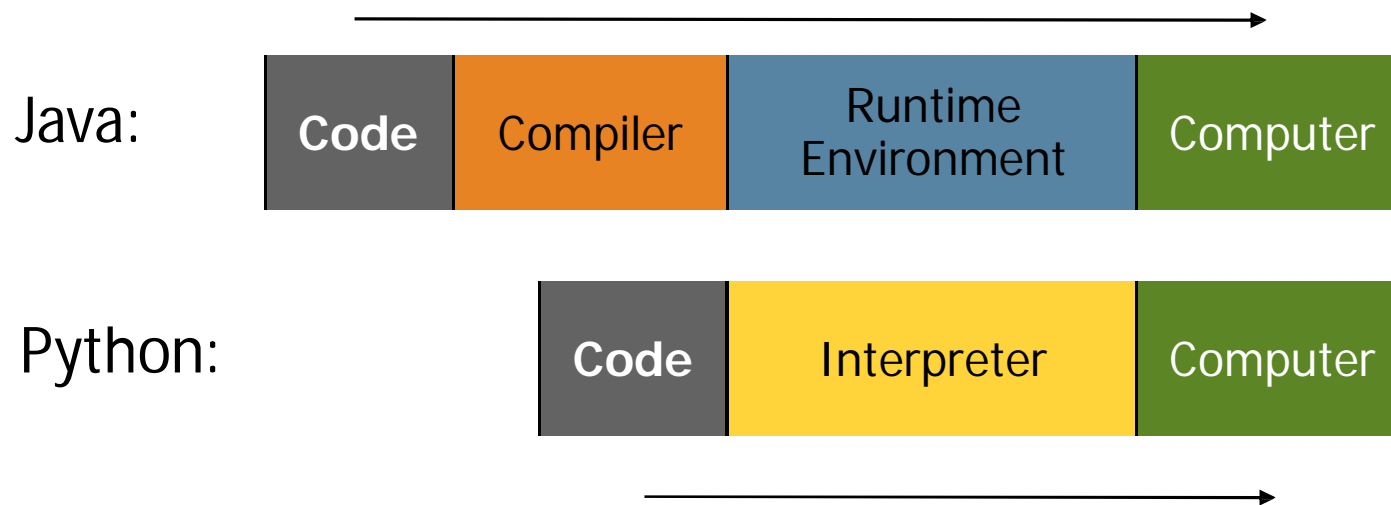
- Here are the simple steps to install Python on
- Unix/Linux machine.
 - Open a Web browser and go to <http://www.python.org/download/>.
 - Follow the link to download zipped source code available for Unix/Linux, download and extract files.
 - Editing the *Modules/Setup* file if you want to customize some options.
 - **run `./configure` script, `make` & `make install`**
- This installs Python at standard location *`/usr/local/bin`* and its libraries at *`/usr/local/lib/pythonXX`* where *XX* is the version of Python.

Windows Installation

- Here are the steps to install Python on Windows machine.
 - Open a Web browser and go to <http://www.python.org/download/>
 - Follow the link for the Windows installer *python-XYZ.msi* file
 - *where XYZ is the version you need to install.*
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Interpreted Language

- **interpreted**
 - Not compiled like Java
 - Code is written and then directly executed by an **interpreter**
 - Type commands into interpreter and see immediate results



What sort of language is Python?

Compiled

Explicitly
compiled
to machine
code

C, C++,
Fortran

Explicitly
compiled
to byte
code

Java, C#

Implicitly
compiled
to byte
code

Python

Interpreted

Purely
interpreted

Shell,
Perl



Simple Programme

- Python does not have a main method like Java
 - The program's main code is just written directly in the file
- Python statements do not end with semicolons

helloJNCCE.py

```
1 Print("Hello, JNCCE!")
```

- Escape sequences such as \" are the same as in Java
- Strings can also start/end with '

Who uses Python?

On-line games

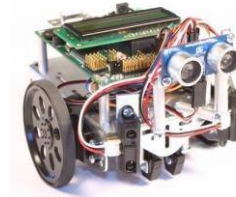
Web services

Applications

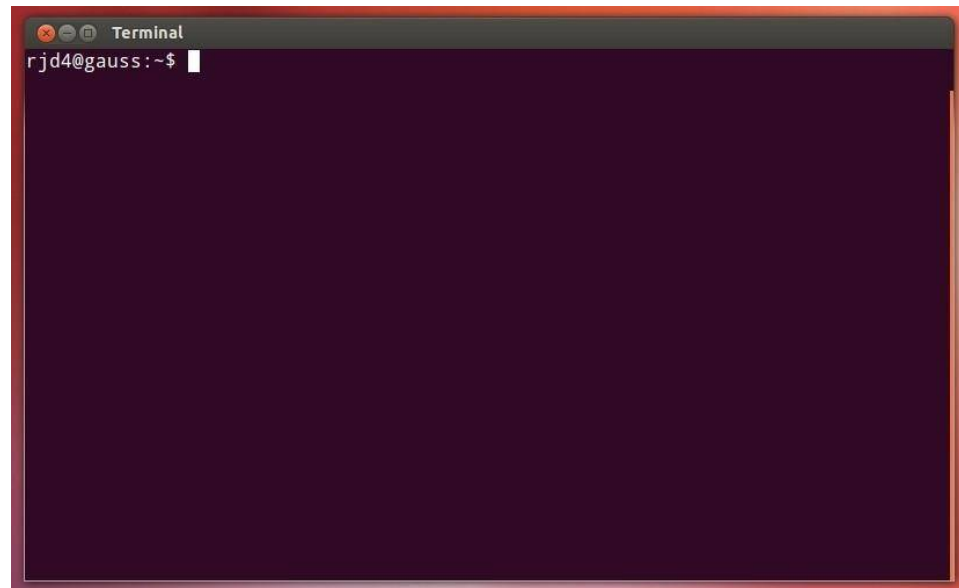
Science

Instrument control

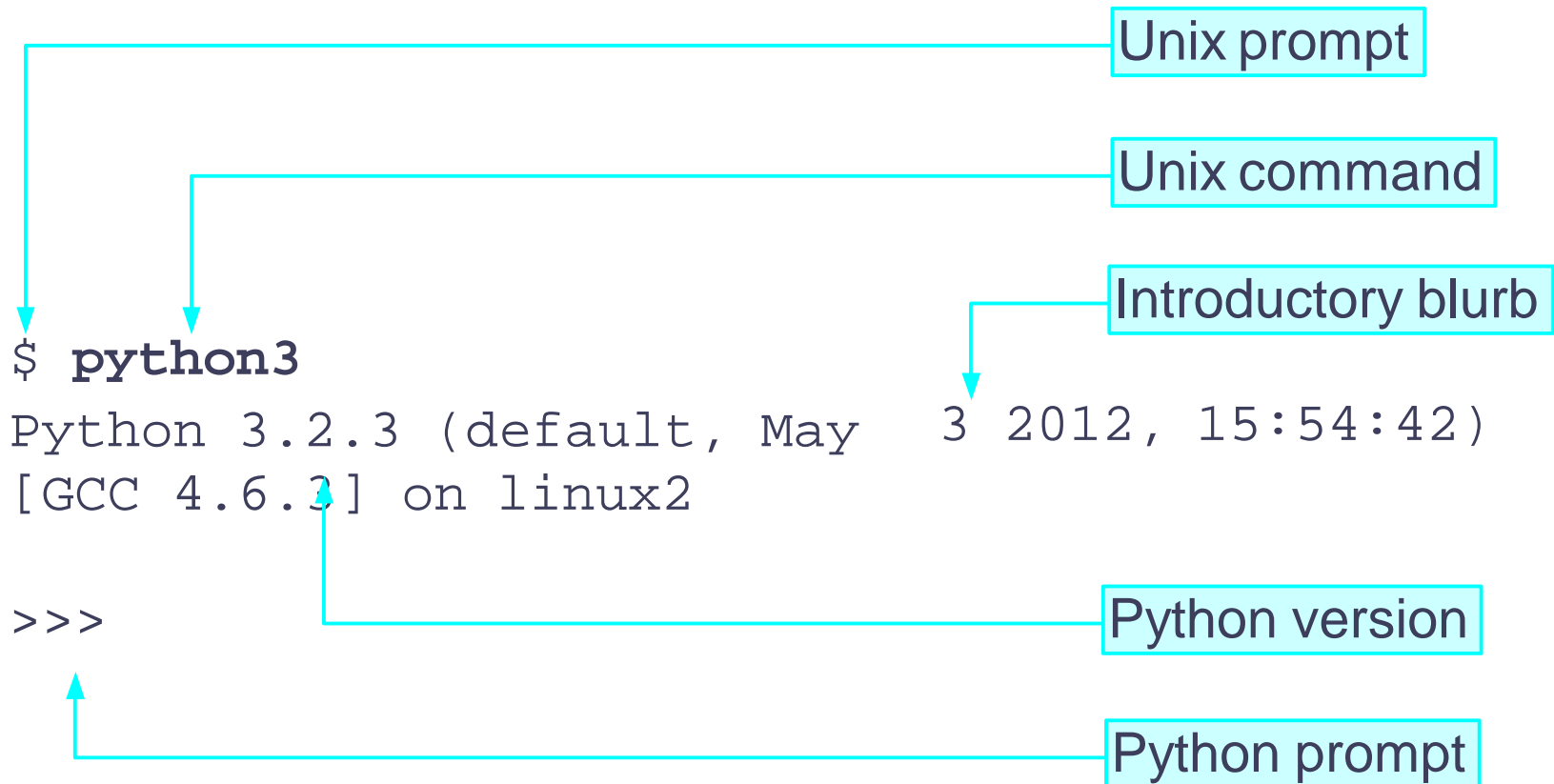
Embedded systems



Running Python — 1



Running Python — 2




Quitting Python

```
>>> exit()
```

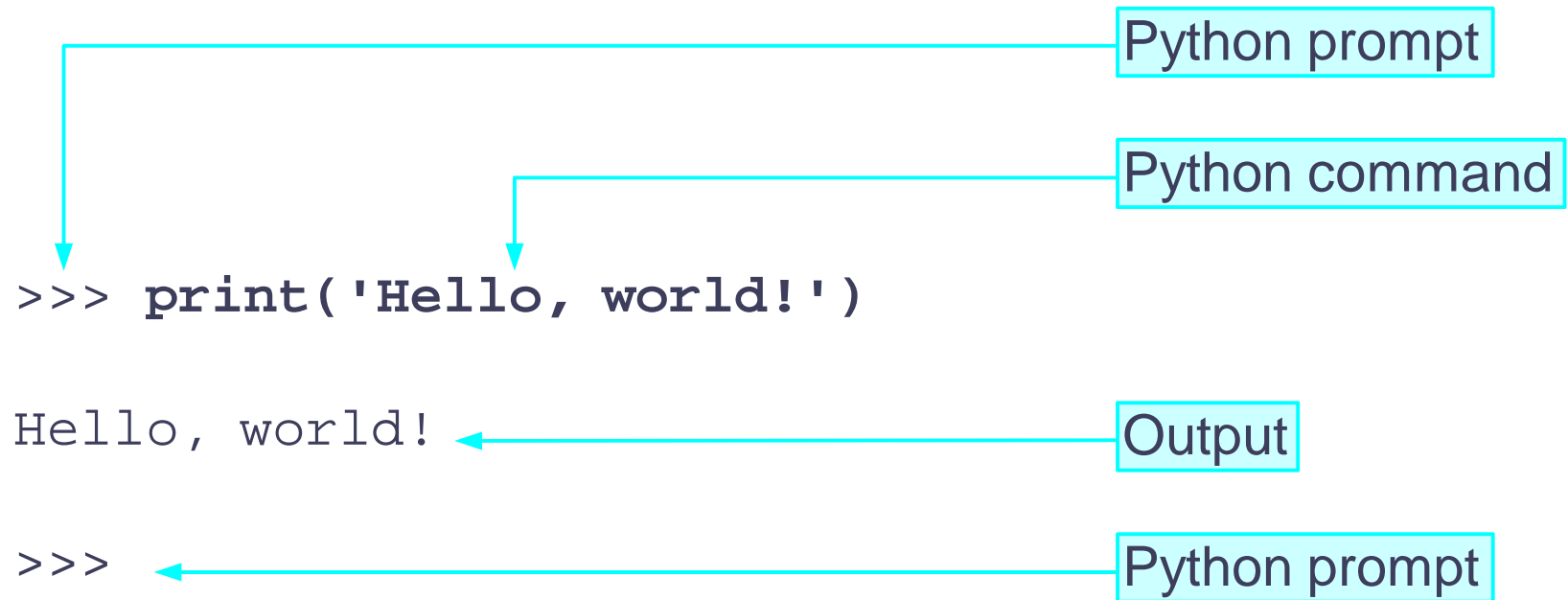
```
>>> quit()
```

```
>>> Ctrl + D
```

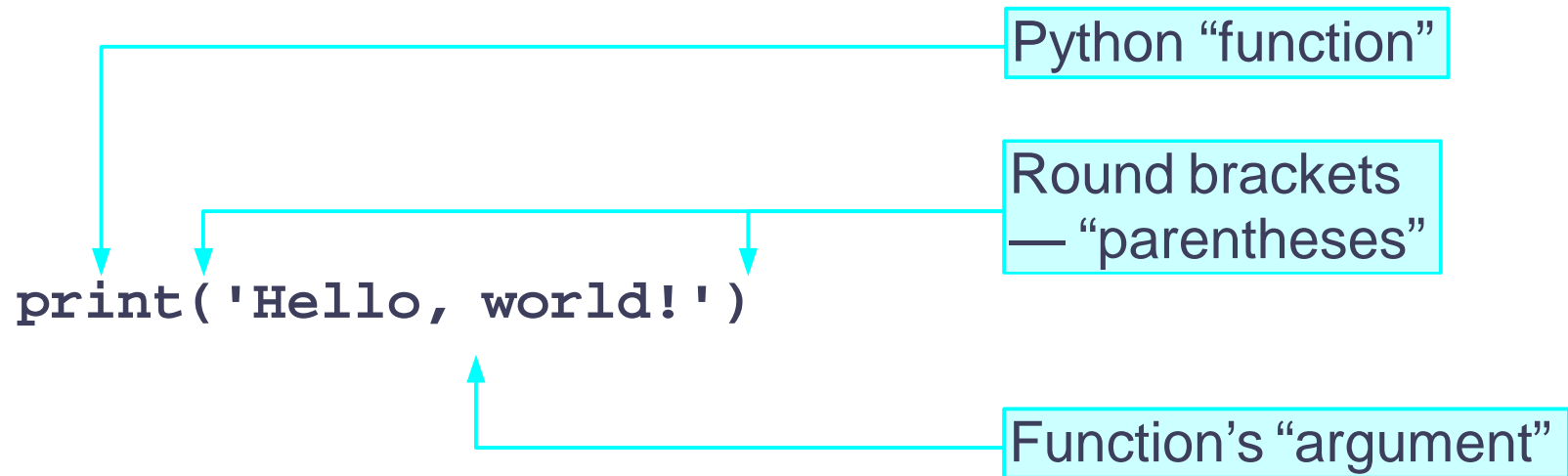


Any one
of these

A first Python command



Python commands



`print` \neq `PRINT`

“Case sensitive”

Expressions

- Python commands are called *statements*.
- *One kind of statement is an expression statement, or expression.*
- Like any programming language, Python can *evaluate basic mathematical* expressions.
- For example, the following expression adds 4 and 13:

```
>>> 4 + 13  
17
```

Type int

- It's not surprising that $4 + 13$ is 17.
- For example, look at what happens when we divide 17 by 10:

```
>>> 17 / 10  
1
```

- You would expect the result to be 1.7, but Python produces 1 instead. This is because every value in Python has a particular *type*, *and the types of values* determine how they behave when they're combined..
- In Python, an expression involving values of a certain type produces a value of that same type. For example, 17 and 10 are integers—in Python, we say they are of type int. When we divide one by the other, the result is also an int.

Type float

- Python has another type called float to represent numbers with fractional parts.

- An expression involving two floats produces a float:

```
>>> 17.0 / 10.0  
1.7
```

- When an expression's operands are an int and a float, Python automatically converts the int to a float.

```
>>> 17.0 / 10  
1.7
```

```
>>> 17 / 10.0  
1.7
```

```
>>> 17 / 10.  
1.7
```

```
>>> 17. / 10  
1.7
```

Operator	Symbol	Example	Result
-	Negation	-5	-5
*	Multiplication	8.5 * 3.5	29.75
/	Division	11 / 3	3
%	Remainder	8.5 % 3.5	1.5
+	Addition	11 + 3	14
-	Subtraction	5 - 19	-14
**	Exponentiation	2 ** 5	32

What Is a Type?

- In computing, a type is a set of values, along with a set of operations that can be performed on those values.
- For example, the type `int` is the values ..., -3, -2, -1, 0, 1, 2, 3, ..., along with the operators `+`, `-`, `*`, `/`, and `%`.
- 84.2 is a member of the set of float values.

- Look at Python's version of the fraction $1/3$

```
>>> 1.0 / 3.0
```

```
0.33333333333333331
```

- What's that 1 doing at the end? Shouldn't it be a 3? The problem is that real computers have a finite amount of memory, which limits how much information they can store about any single number.

Operator Precedence

- Let's convert Fahrenheit to Celsius. To do this, we subtract 32 from the temperature in Fahrenheit and then multiply by 5 / 9:

```
>>> 212 - 32.0 * 5.0 / 9.0  
194.22222222222223
```

- Python claims the result is 194.222222222222232 degrees Celsius when in fact it should be 100. The problem is that * and / have higher *precedence* than - .
- This means that what we actually calculated was $212 - ((32.0 * 5.0) / 9.0)$.
- We can alter the order of precedence by putting parentheses around parts of the expression

```
>>> (212 - 32.0) * 5.0 / 9.0  
100.0
```

Variables and the Assignment Statement

- *Variable* is just a name that has a value associated with it.
- Variables' names can use letters, digits, and the underscore symbol.
- For example, X, species5618, and degrees_celsius are all allowed, but 777 isn't.
- You create a new variable simply by giving it a value:

```
>>> degrees_celsius = 26.0
```
- This statement is called an *assignment statement*; we say that *degrees_celsius* is assigned the value 26.0.
- An assignment statement is executed as follows:
 1. Evaluate the expression on the right of the = sign.
 2. Store that value with the variable on the left of the = sign.

- In the diagram below, we can see the *memory model for the result of the* assignment statement.

`degrees_celsius` → 26.0

- Once a variable has been created, we can use its value in other calculations.

```
>>> 100 - degrees_celsius
74.0
```

- Whenever the variable's name is used in an expression, Python uses the variable's value in the calculation.

- This means that we can create new variables from old ones:

```
>>> difference = 100 - degrees_celsius
```

- Typing in the name of a variable on its own makes Python display its value:

```
>>> difference
74.0
```

- Variables are called variables because their values can change as the program executes.

```
>>> difference = 100 - 15.5
```

```
>>> difference
```

```
84.5
```

- This does *not change the results of any calculations done with that variable* before its value was changed:

```
>>> difference = 20
```

```
>>> double = 2 * difference
```

```
>>> double
```

```
40
```

```
>>> difference = 5
```

```
>>> double
```

```
40
```

- We can even use a variable on both sides of an assignment statement:

```
>>> number = 3
```

```
>>> number
```

```
3
```

```
>>> number = 2 * number
```

```
>>> number
```

```
6
```

```
>>> number = number * number
```

```
>>> number
```

```
36
```

- When a statement like `number = 2 * number` is evaluated, Python does the following:
 1. Gets the value currently associated with `number`
 2. Multiplies it by 2 to create a new value
 3. Assigns that value to `number`

Combined Operators

- In the previous example, the variable number appeared on both sides of the assignment statement.
- This is so common that Python provides a shorthand notation for this operation:

```
>>> number = 100
```

```
>>> number -= 80
```

```
>>> number
```

```
20
```

- Here is how a *combined operator is evaluated*:
 1. Evaluate the expression to the right of the = sign.
 2. Apply the operator attached to the = sign to the variable and the result of the expression.
 3. Assign the result to the variable to the left of the = sign.

```
>>> difference = 20
```

difference → 20

```
>>> double = 2 * difference
```

difference → 20

double → 40

```
>>> difference = 5
```

difference → 5

double → 40

Changing a variable's value

How Python Tells You Something Went Wrong

- There are two kinds of errors in Python:
- *Syntax errors* : happens when you type something that isn't valid Python code.
- *Semantic errors* : happens when you tell Python to do something that it just can't do, like divide a number by zero or try to use a variable that doesn't exist.
- Here is what happens when we try to use a variable that hasn't been created yet:

```
>>> 3 + moogah
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

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

- The first two lines aren't much use right now, though they'll be indispensable when we start writing longer programs.
- The last line is the one that tells us what went wrong: the name `moogah` wasn't recognized.
- The rules governing what is and isn't legal in a programming language are called its *syntax*.
- *The message tells us that we violated Python's syntax rules.*

A Single Statement That Spans Multiple Lines

- In order to split up a statement into more than one line, you need to do one of two things:
 1. Make sure your line break occurs inside parentheses, or
 2. Use the line-continuation character, which is a backslash, \.
- Note that the line-continuation character is a backslash (\), not the division symbol (/).
- Here are examples of both:

```
>>> (2 +  
... 3)  
5
```

```
>>> 2 + \  
... 3  
5
```

- Notice how we don't get a `SyntaxError`.
- Each triple-dot prompt in our examples indicates that we are in the middle of entering an expression; we use them to make the code line up nicely.
- Here is one more example: let's say we're baking cookies.
- The authors live in Canada, which uses Celsius, but we own cookbooks that use Fahrenheit.
- So how long it will take to preheat our oven? Here are our facts:
 - The room temperature is 20 degrees Celsius.
 - Our oven controls use Celsius, and the oven heats up at 20 degrees per minute.
 - Our cookbook uses Fahrenheit, and it says to preheat the oven to 350 degrees.

- We can convert t degrees Fahrenheit to t degrees Celsius like this:
 $(t - 32) * 5 / 9$.

- Let's use this information to try to solve our problem.

```
>>> room_temperature_c = 20
```

```
>>> cooking_temperature_f = 350
```

```
>>> oven_heating_rate_c = 20
```

```
>>> oven_heating_time = (
```

```
... ((cooking_temperature_f - 32) * 5 / 9) - room_temperature_c) / \
```

```
... oven_heating_rate_c
```

```
>>> oven_heating_time
```

```
7.833333333333333
```

Designing and Using Functions

- Mathematicians create *functions to make calculations* easy to reuse and to make other calculations easier to read because they can use those functions instead of repeatedly writing out equations.
- Programmers do this too.

Functions That Python Provides

- Python comes with many *built-in functions that perform common operations*.

- One example is `abs`, which produces the absolute value of a number:

```
>>> abs(-9)
```

```
9
```

```
>>> abs(3.3)
```

```
3.3
```

- Each of these statements is a *function call*.
- The general form of a function call is as follows:
«function_name»(«arguments»)

- Here are the rules to executing a function call:
 1. Evaluate each argument one at a time, working from left to right.
 2. Pass the resulting values into the function.
 3. Execute the function. When the function call finishes, it produces a value.

- As function calls produce values, they can be used in expressions:

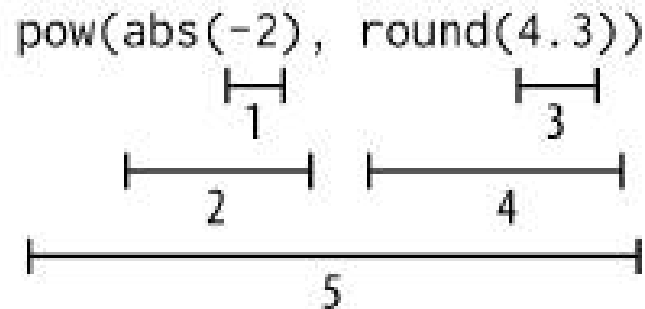
```
>>> abs(-7) + abs(3.3)  
10.3
```

- We can also use function calls as arguments to other functions:

```
>>> pow(abs(-2), round(4.3))  
16
```

```
>>> pow(abs(-2), round(4.3))  
16
```

- Python sees the call on pow and starts by evaluating the arguments from left to right.
- The first argument is a call on function abs, so Python executes it.
- abs(-2) produces 2, so that's the first value for the call on pow.
- Then Python executes round(4.3), which produces 4.



- Some of the most useful built-in functions are ones that convert from one type to another.

```
>>> int(34.6)
```

```
34
```

```
>>> int(-4.3)
```

```
-4
```

```
>>> float(21)
```

```
21.0
```

- In this example, we see that when a floating-point number is converted to an integer, it is truncated, not rounded.
- If you're not sure what a function does, try calling built-in function `help`, which shows documentation for any function:

```
>>> help(abs)
```

```
Help on built-in function abs in module builtins:
```

```
abs(...)
```

```
    abs(number) -> number
```

```
    Return the absolute value of the argument.
```

- Another built-in function is `round`, which rounds a floating-point number to the nearest integer:

```
>>> round(3.8)
```

```
4
```

```
>>> round(3.3)
```

```
3
```

```
>>> round(3.5)
```

```
4
```

```
>>> round(-3.3)
```

```
-3
```

```
>>> round(-3.5)
```

```
-4
```

- Function round can be called with one or two arguments.
- If called with one, it rounds to the nearest integer.
- If called with two, it rounds to a floating-point number, where the second argument indicates the precision:

```
>>> round(3.141592653, 2)  
3.14
```

- Let's explore built-in function pow by starting with its help documentation:

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

```
pow(x, y[, z]) -> number
```

With two arguments, equivalent to $x^{**}y$. With three arguments, equivalent to $(x^{**}y) \% z$, but may be more efficient (e.g. for longs).

- This shows that function pow can be called with either two or three arguments.
- The English description mentions that when called with two arguments it is equivalent to $x^{**}y$.

```
>>> pow(2, 4)
```

```
16
```

- This call calculates 2^4 .
- How about with three arguments?

```
>>> pow(2, 4, 3)
```

```
1
```

- We know that 2^4 is 16, and evaluation of $16 \% 3$ produces 1.

Tracing Function Calls in the Memory Model

```
>>> def f(x):  
... x = 2 * x  
... return x  
...  
>>> x = 1  
>>> x = f(x + 1) + f(x + 2)
```

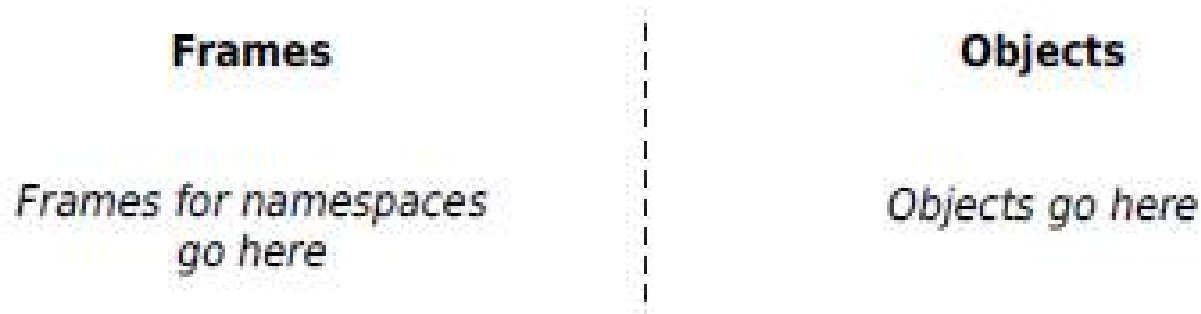
- Are all the x's the same variable?
- Does Python make a new x for each assignment? For each function call?
- For each function definition?

- Here's the answer: whenever Python executes a function call, it creates a *namespace (literally, a space for names) in which to store local variables for that call.*
- You can think of a namespace as a scrap piece of paper: Python writes down the local variables on that piece of paper, keeps track of them as long as the function is being executed, and throws that paper away when the function returns.
- Separately, Python keeps another namespace for variables created in the shell.
- Means the x that is a parameter of function f is a different variable than the x in the shell.

- Let's refine our rules for executing a function call to include this namespace creation:

1. Evaluate the arguments left to right.
2. Create a namespace to hold the function call's local variables, including the parameters.
3. Pass the resulting argument values into the function by assigning them to the parameters.
4. Execute the function body.

- In our memory model, we will draw a separate box for each namespace to indicate that the variables inside it are in a separate area of computer memory.
- The programming world calls this box a *frame*.
- *We separate* the frames from the objects by a vertical dotted line:



- Let's trace that confusing code. At the beginning, no variables have been created; Python is about to execute the function definition.

- We have indicated this with an arrow:

```
>>> def f(x):
```

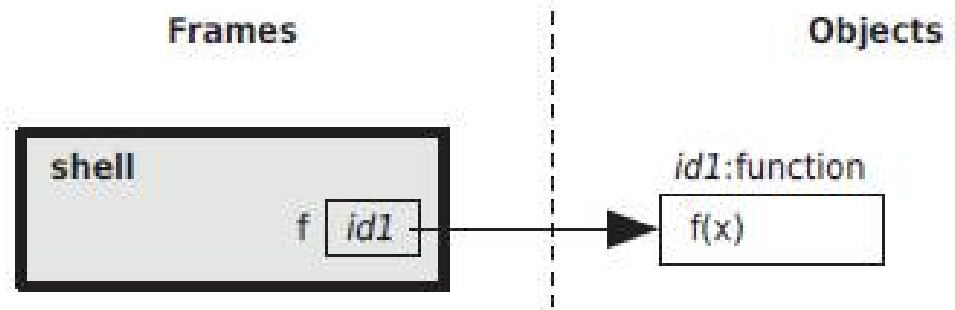
```
... x = 2 * x
```

```
... return x
```

```
...
```

```
>>> x = 1
```

```
>>> x = f(x + 1) + f(x + 2)
```



- When Python executes that function definition, it creates a variable `f` in the frame for the shell's namespace plus a function object.

- Now we are about to execute the **first assignment** to x in the shell.

```
>>> def f(x):
```

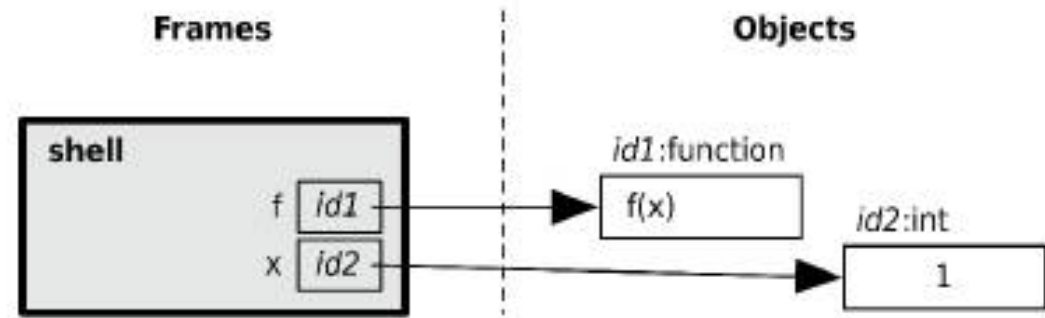
```
... x = 2 * x
```

```
... return x
```

```
...
```

```
>>> x = 1
```

```
>>> x = f(x + 1) + f(x + 2)
```



- Once that assignment happens, both f and x are in the frame for the shell.
- Now we are about to execute the second assignment to x in the shell:

```
>>> def f(x):
```

```
... x = 2 * x
```

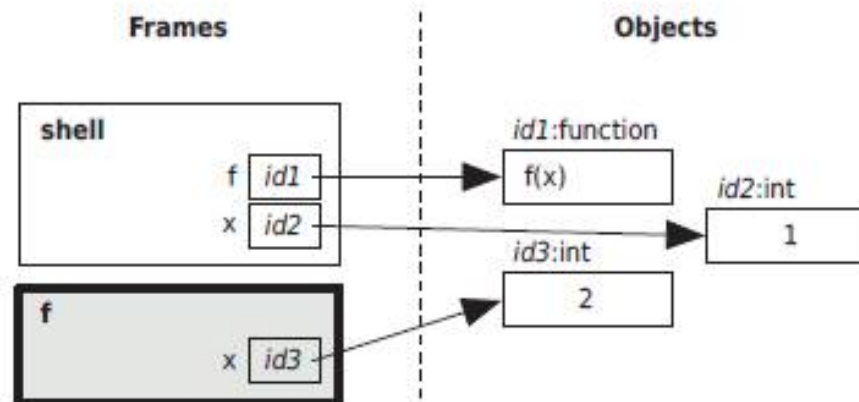
```
...
```

```
>>> x = 1
```

```
>>> x = f(x + 1) + f(x + 2)
```

- We first evaluate the expression on the right of the $=$, which is $f(x + 1) + f(x + 2)$.
- Python evaluates the left function call first: $f(x + 1)$.
- Python evaluates the argument, $x + 1$.
- In order to find the value for x , Python looks in the current frame.
- The current frame is the frame for the shell, and its variable x refers to 1, so $x + 1$ evaluates to 2.
- After evaluating the argument to f , the next step is to create a namespace for the function call.

- We draw a frame, write in parameter x, and assign 2 to that parameter:



- There are two variables called x, and they refer to different values.
- Python will always look in the current frame, which we will draw with a thicker border.

- We are now about to execute the **first statement** of function f:

```
>>> def f(x):
```

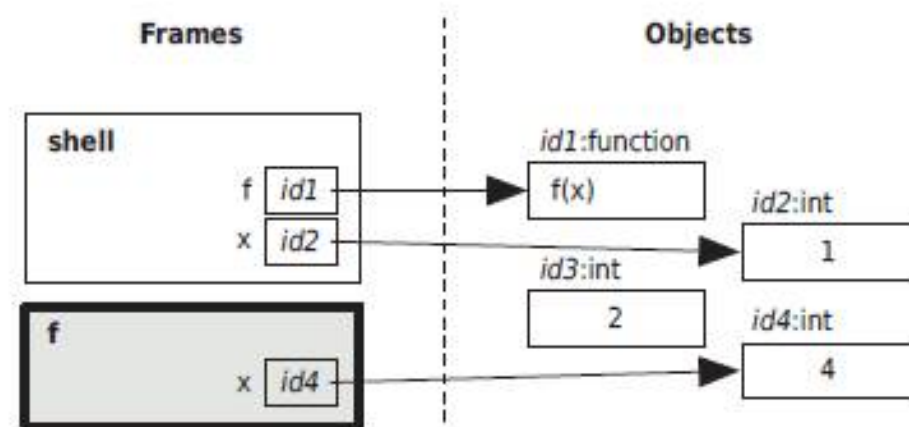
```
>>>   x = 2 * x
```

```
...   return x
```

```
...
```

```
>>> x = 1
```

```
>>> x = f(x + 1) + f(x + 2)
```

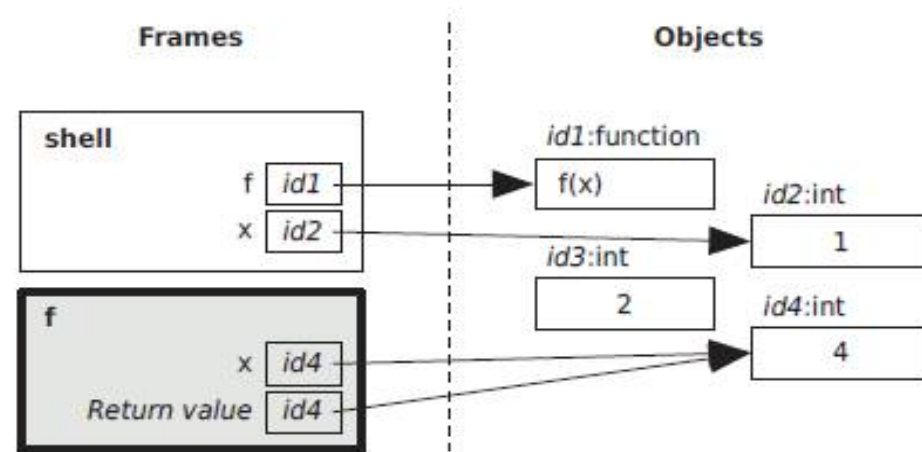


- `x = 2 * x` is an assignment statement.
- The right side is the expression `2 * x`.
- Python looks up the value of `x` in the current frame and finds 2, so that expression evaluates to 4.
- Python finishes executing that assignment statement by making `x` refer to that 4.

- We are now about to execute the **second statement** of function f:

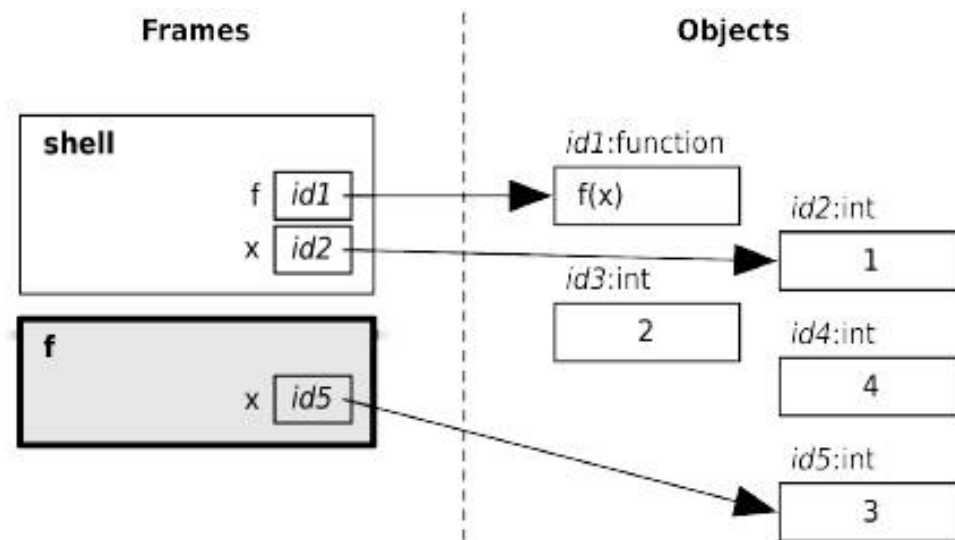
```
>>> def f(x):
... x = 2 * x
➤ ... return x
...
```

```
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```



- This is a return statement, so we evaluate the expression, which is simply x.
- Python looks up the value for x in the current frame and finds 4, so that is the return value.
- When the function returns, Python comes back to this expression: $f(x + 1) + f(x + 2)$.
- Python just finished executing $f(x + 1)$, which produced the value 4.
- It then executes the right function call: $f(x + 2)$.

- Following the rules for executing a function call, Python evaluates the argument, $x + 2$.
- In order to find the value for x , Python looks in the current frame.
- The call on function f has returned, so that frame is erased: the only frame left is the frame for the shell, and its variable x still refers to 1, so $x + 2$ evaluates to 3.
- Now we have evaluated the argument to f . The next step is to create a namespace for the function call.
- We draw a frame, write in the parameter x , and assign 3 to that parameter:



We are now about to execute the first statement of function f:

```
>>> def f(x):
```

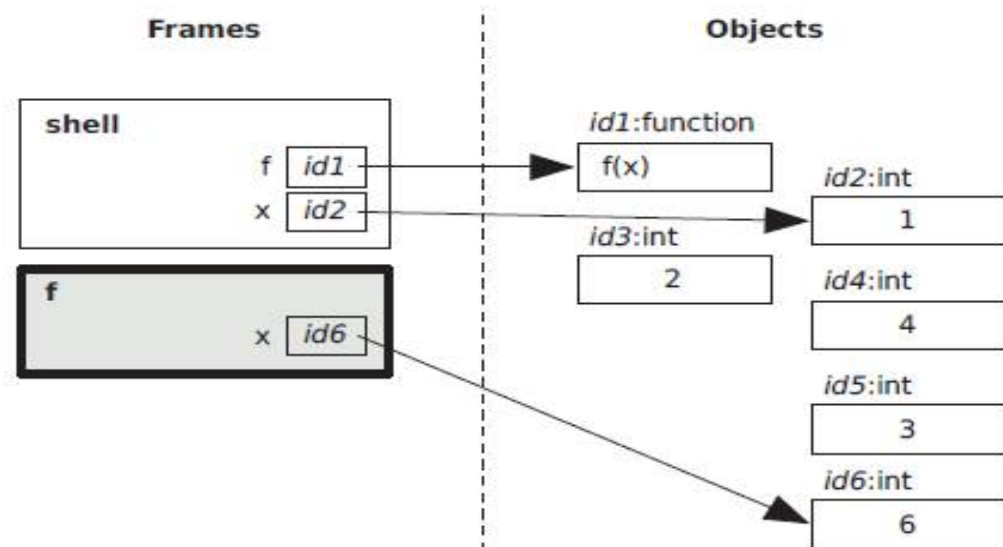
```
>>   x = 2 * x
```

```
...   return x
```

```
...
```

```
>>> x = 1
```

```
>>> x = f(x + 1) + f(x + 2)
```



- `x = 2 * x` is an assignment statement.
- The right side is the expression `2 * x`.
- Python looks up the value of `x` in the current frame and finds 3, so that expression evaluates to 6.
- Python finished executing that assignment statement by making `x` refer to that 6.

- We are now about to execute the second statement of function f:

```
>>> def f(x):
```

```
...
```

```
...
```

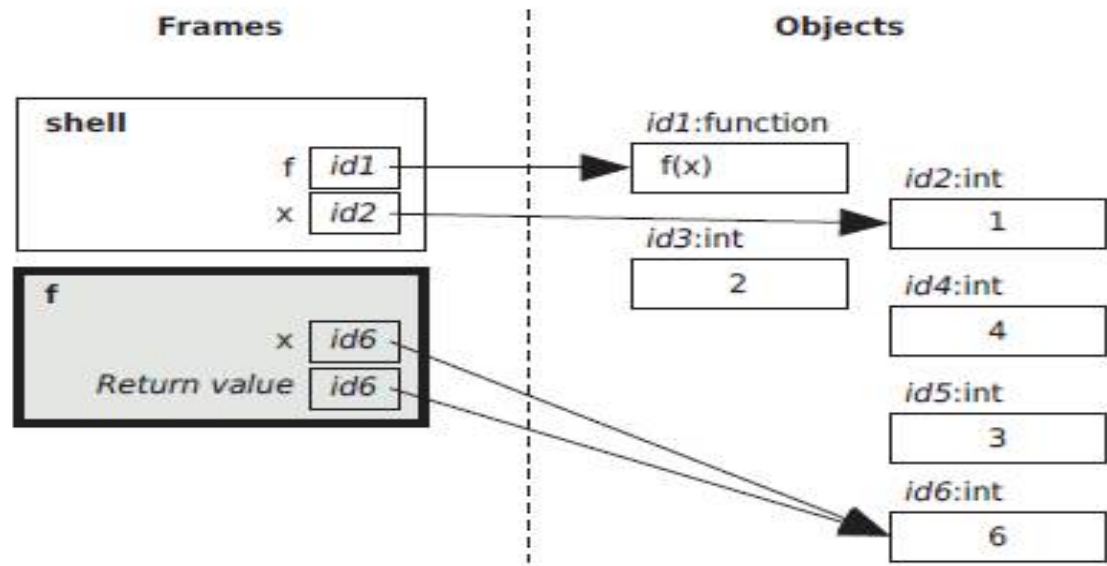
```
x = 2 * x
```

```
> return x
```

```
...
```

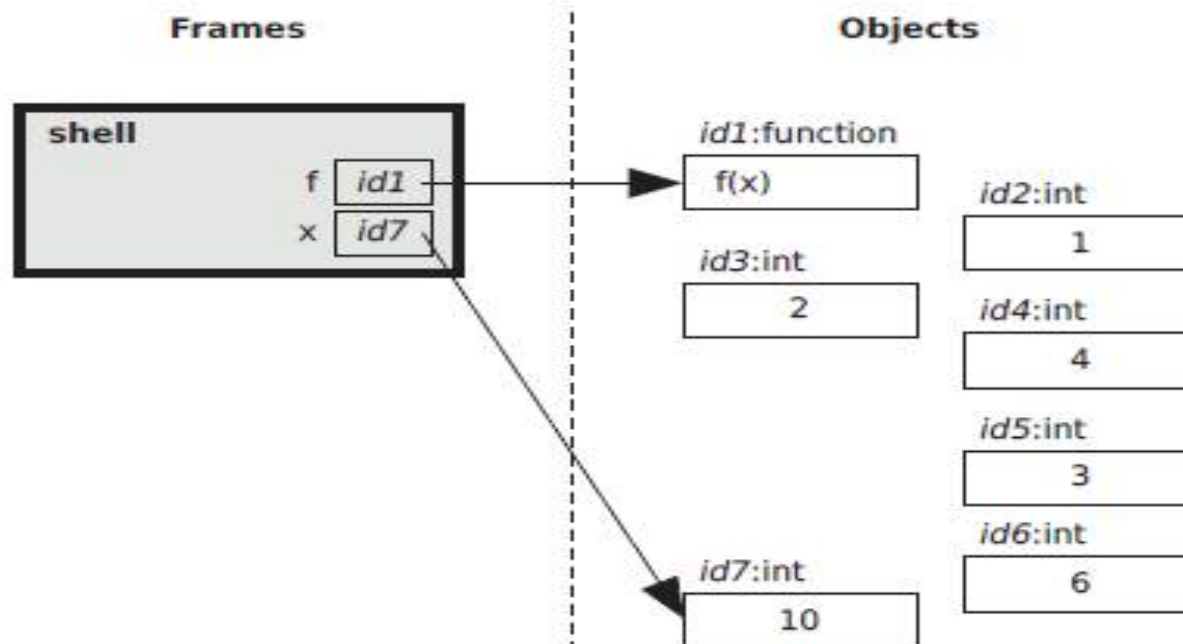
```
>>> x = 1
```

```
>>> x = f(x + 1) + f(x + 2)
```



- This is a return statement, so we evaluate the expression, which is simply `x`.
- Python looks up the value for `x` in the current frame and finds 6, so that is the return value.

- When the function returns, Python comes back to this expression:
 $f(x + 1) + f(x + 2)$.
- Python just finished executing $f(x + 2)$, which produced the value 6.
- Both function calls have been executed, so Python applies the + operator to 4 and 6, giving us 10.
- We have now evaluated the right side of the assignment statement; Python completes it by making the variable on the left side, x, refer to 10.



Omitting a Return Statement: None

- If you don't have a return statement in a function, nothing is produced:

```
>>> def f(x):  
... x = 2 * x
```

```
...
```

```
>>> res = f(3)
```

```
>>> res
```

```
>>>
```

- That can't be right—if res doesn't have a value, shouldn't we get a NameError? Let's poke a little more:

```
>>> print(res)
```

```
None
```

```
>>> id(res)
```

```
1756120
```

- Variable `res` has a value: it's `None`! And `None` has a memory address.
- If you don't have a `return` statement in your function, your function will return `None`.
- You can return `None` yourself if you like:

```
>>> def f(x):  
...     x = 2 * x  
...     return None  
...  
>>> print(f(3))  
None
```

- The value `None` is used to signal the absence of a value.

Working with Text

Creating Strings of Characters

- In Python, text is represented as a *string*, which is a sequence of characters.
- We indicate that a value is a string by putting either single or double quotes around it.
- Single and double quotes are equivalent except for strings that contain quotes.

- Here are two examples:

The opening and closing quotes must match:

```
>>> 'Aristotle'
```

```
'Aristotle'
```

```
>>> "Isaac Newton"
```

```
'Isaac Newton'
```

```
>>> 'Charles Darwin"
```

```
File "<stdin>", line 1
```

```
'Charles Darwin'
```

^

SyntaxError: EOL while scanning string literal

- Strings can contain any number of characters, limited only by computer memory.
- The shortest string is the *empty string*, containing no characters at all:

```
>>> ''  
''
```

```
>>> ""  
''
```

Operations on Strings

- Python has a built-in function, `len`, that returns the number of characters between the opening and closing quotes:

```
>>> len('Albert Einstein')
```

```
15
```

```
>>> len('123!')
```

```
4
```

```
>>> len(' ')
```

```
1
```

```
>>> len("")
```

```
0
```

- We can add two strings using the `+` operator, which produces a new string containing the same characters as in the two operands:

```
>>> 'Albert' + ' Einstein'
```

```
'Albert Einstein'
```

- When `+` has two string operands, it is referred to as the *concatenation operator*.

- Adding an empty string to another string produces a new string that is just like the nonempty operand:

```
>>> "Alan Turing" + "
```

```
'Alan Turing'
```

```
>>> "" + 'Grace Hopper'
```

```
'Grace Hopper'
```

- Can operator + be applied to a string and a numeric value? If so, would addition or concatenation occur?

```
>>> 'NH' + 3
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: Can't convert 'int' object to str implicitly
```

- Here, Python took exception to our attempts to combine values of different data types because it didn't know which version of + we want: the one that adds numbers or the one that concatenates strings.

```
>>> 9 + ' planets'
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- Here, because Python saw a 9 first, it expected the second operand to also be numeric. The order of the operands affects the error message.
- The concatenation operator must be applied to two strings.
- If you want to join a string with a number, function str can be applied to the number to get a string representation of it, and then the concatenation can be done:

```
>>> 'Four score and ' + str(7) + ' years ago'
```

```
'Four score and 7 years ago'
```

- Function `int` can be applied to a string whose contents look like an integer, and `float` can be applied to a string whose contents are numeric:

```
>>> int('0')
```

```
0
```

```
>>> int("11")
```

```
11
```

```
>>> int('-324')
```

```
-324
```

```
>>> float('-324')
```

```
-324.0
```

```
>>> float("56.34")
```

```
56.34
```

```
>>> int('a')
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ValueError: invalid literal for int() with base 10: 'a'
```

```
>>> float('b')
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ValueError: could not convert string to float: 'b'
```

- In addition to +, len, int, and float, operator * can be applied to strings.
- A string can be repeated using operator * and an integer, like this:

```
>>> 'AT' * 5
'ATATATATAT'
>>> 4 * '-'
'----'
```

- If the integer is less than or equal to zero, the operator yields the empty string (a string containing no characters):

```
>>> 'GC' * 0
''

>>> 'TATATATA' * -3
''
```

- Strings are values, so you can assign a string to a variable.
- Also, operations on strings can be applied to those variables:

```
>>> sequence = 'ATTGTCCCCC'
```

```
>>> len(sequence)
```

```
10
```

```
>>> new_sequence = sequence + 'GGCCTCCTGC'
```

```
>>> new_sequence
```

```
'ATTGTCCCCCGGCCTCCTGC'
```

```
>>> new_sequence * 2
```

```
'ATTGTCCCCCGGCCTCCTGCATTGTCCCCCGGCCTCCTGC'
```

Using Special Characters in Strings

- Suppose you want to put a single quote inside a string. If you write it directly, an error occurs:

```
>>> 'that's not going to work'
```

```
File "<stdin>", line 1
```

```
'that's not going to work'
```

```
      ^
```

```
SyntaxError: invalid syntax
```

- When Python encounters the second quote—the one that is intended to be part of the string—it thinks the string is ended.
- Then it doesn't know what to do with the text that comes after the second quote.

- One simple way to fix this is to use double quotes around the string.

```
>>> "that's better"
```

```
"that's better"
```

```
>>> 'She said, "That is better."'
```

```
'She said, "That is better."'
```

- If you need to put a double quote in a string, you can use single quotes around the string.
- But what if you want to put both kinds of quote in one string? You could do this:

```
>>> 'She said, "That" + "'" + 's hard to read.'
```

```
'She said, "That\'s hard to read.'
```

- The backslash is called an *escape character*, and the combination of the backslash and the single quote is called an *escape sequence*.

- The escape sequence `\'` is indicated using two symbols, but those two symbols represent a single character.
- The length of an escape sequence is one:

```
>>> len('\')
```

```
1
```

```
>>> len('it\'s')
```

```
4
```

- Here are some common escape sequences:

Escape Sequence	Description
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\t</code>	Tab
<code>\n</code>	Newline
<code>\r</code>	Carriage return

Creating a Multiline String

- If you create a string using single or double quotes, the whole string must fit onto a single line.
- Here's what happens when you try to stretch a string across multiple lines:

```
>>> 'one
```

```
File "<stdin>", line 1
```

```
'one
```

```
^
```

SyntaxError: EOL while scanning string literal

- To span multiple lines, put three single quotes or three double quotes around the string instead of one of each.

```
>>> '''one
```

```
... two
```

```
... three'''
```

```
'one\ntwo\nthree'
```

Printing Information

- We will use `print` to print messages to the users of our program.
- Those messages may include the values that expressions produce and the values that the variables refer to.

```
>>> print(1 + 1)
```

```
2
```

```
>>> print("The Latin 'Oryctolagus cuniculus' means 'domestic  
rabbit'.")
```

```
The Latin 'Oryctolagus cuniculus' means 'domestic rabbit'.
```

- Function `print` doesn't allow any styling of the output: no colors, no italics, no boldface. All output is plain text.
- The second ex. strips off the quotes around the string and shows us the string's contents rather than its representation.

```
>>> print('one\ttwo\nthree\tfour')
```

```
one    two
three  four
```

- The example above shows how the tab character `\t` can be used to lay values out in columns.
- When a multiline string is printed, those `\n` sequences are displayed as new lines:

```
>>> numbers = "one
... two
... three"
```

```
>>> numbers
```

```
'one\ntwo\nthree'
```

```
>>> print(numbers)
```

```
one
two
three
```

- Function `print` takes a comma-separated list of values to print and prints the values with a single space between them and a newline after the last value:

```
>>> print(1, 2, 3)
```

```
1 2 3
```

```
>>>
```

- When called with no arguments, `print` ends the current line, advancing to the next one:

```
>>> print()
```

```
>>>
```

- Function `print` can print values of any type, and it can even print values of different types in the same function call:

```
>>> print(1, 'two', 'three', 4.0)
```

```
1 two three 4.0
```

- We separate each value with a comma and a space instead of just a space by including `sep=', '` as an argument:

```
>>> print('a', 'b', 'c') # The separator is a space by default
```

```
a b c
```

```
>>> print('a', 'b', 'c', sep=', ')
```

```
a, b, c
```

- Use the keyword argument `end=""` to tell Python to end with an empty string instead of a new line:

```
>>> print('a', 'b', 'c', sep=', ', end="")
```

```
a, b, c>>>
```

- `end=""` is used only in programs, not in the shell.

Getting Information from the Keyboard

- Another built-in function that you will find useful is `input`, which reads a single line of text from the keyboard.
- It returns whatever the user enters as a string, even if it looks like a number:

```
>>> species = input()
```

```
Homo sapiens
```

```
>>> species
```

```
'Homo sapiens'
```

```
>>> population = input()
```

```
6973738433
```

```
>>> population
```

```
'6973738433'
```

```
>>> type(population)
```

```
<class 'str'>
```


- If you are expecting the user to enter a number, you must use `int` or `float` to get an integer or a floating-point representation of the string:

```
>>> population = input()
```

```
6973738433
```

```
>>> population
```

```
'6973738433'
```

```
>>> population = int(population)
```

```
>>> population
```

```
6973738433
```

```
>>> population = population + 1
```

```
>>> population
```

```
6973738434
```

- This time function `int` is called on the result of the call to `input`.

```
>>> population = int(input())
```

```
6973738433
```

```
>>> population = population + 1
```

```
6973738434
```

- Input can be given a string argument, which is used to prompt the user for input (notice the space at the end of our prompt):

```
>>> species = input("Please enter a species: ")
```

```
Please enter a species: Python curtus
```

```
>>> print(species)
```

```
Python curtus
```