Space Exe Python Workshop 2017

Sam Morrell
@smorrell
smorrell@astro.ex.ac.uk

Introduction

- **There are many different pieces of software for analysing data:**
  - Octave / MATLAB
  - IDL
  - Spreadsheet
  - C / Fortran
  - Scripting

Why Should I Try Python?

- Rule No. 1 – Use the Right Tool for the Right Job.
- Same Code. Most Platforms.
- It's easy to get running and use. Ideal for Prototyping.
- Incredibly Rich Ecosystem of Packages.
- Very Easy to Pick Up.
- Plots!!!

Why You Shouldn't Use Python

- It's So Incredibly Slow.
- Requires the Python Runtime.
- The Language Is in Constant Flux.
- It's Only as Strong as the Community.

Getting Started

- **We Will Use Python 3.**
- **First Install Anaconda. It Runs on Windows, macOS and Linux.**
- **Download / Clone My GitHub Repo.**
  - https://github.com/sammorrell/space-exe-python

Indentation Is Important!

C

```c
if (1) {
    run_this_code();
}

for(int i = 0; i < length(array); i++) {
    printf("%lg\n", array[i]);
}
```

```c
if (1) {
run_this_code();
}

for(int i = 0; i < length(array); i++) {
printf("%lg\n", array[i]);
}
```

Python

```python
if True:
    run_this_code()

for r in array:
    print("{}".format(r))
```

```python
if True:
run_this_code()

for r in array:
print("{}".format(r))
```

Dynamic Typing

C

```c
typedef int bool;
#define true 1
#define false 0
space_exe_is_awesome = true;

int meaning_of_universe = 42;
int negative_integer = -1234;

double pi = 3.14159265;
double negative = -1.68394;
double c = 3.0E8;

char[14] greeting = "Hello, world.";
```

Python

```python
space_exe_is_awesome = True

meaning_of_universe = 42
negative_integer = -1234

pi = 3.14159265
negative = -1.68394
c = 3.0E8

greeting = 'Hello, world.'
```

Python Basics

Assignments and Arithmetic Operators

```
import math
mass = 1
area = 1

a = 1 # m / s^2
b = 2.3
c = 3.0E8 # m / s
d = c * b
e = mass * c ** 2.0 # J
f = mass * a # N
g = 9.81 # m / s
h = 6.67E-34
i = math.sqrt(-1.0)
j = s + l
k = 1.38E-23
l = 5.32 # m
m = d * f # Nm
n = c / 2.25E8
o = True
p = f / area # Pa
```

- Programming is about giving an input, performing a task, and outputting the value.
- Assignments allow the output to be stored in the code.

Assignments and Arithmetic Operators

```
import math
mass = 1
area = 1

a = 1 # m / s^2
b = 2.3
c = 3.0E8 # m / s
d = c * b
e = mass * c ** 2.0 # J
f = mass * a # N
g = 9.81 # m / s
h = 6.67E-34
i = math.sqrt(-1.0)
j = s + l
k = 1.38E-23
l = 5.32 # m
m = d * f # Nm
n = c / 2.25E8
o = True
p = f / area # Pa
```

	Operator	Name	Purpose
3	+	Addition	Add together two operands.
3	-	Subtraction	Subtract the right-hand operand from the left-hand operand.
2	*	Multiplication	Used to multiply two operands together.
2	/	Division	Divides the left-hand operand by the right-hand operand.
2	%	Modulus	Divides the left hand operand by right hand operand and returns the remainder
1	**	Exponent	Raises the left-hand operand to the power of the right-hand operand.

Comparison Operators

```
1 == 2 # False
1 != 2 # True
1 > 2 # False
1 < 2 # True
3 > 2 # True
2 > 2 # False
2 >= 2 # True
1 < 2 # True
2 < 2 # False
2 <= 2 #True
```

	Operator	Name	Purpose
2	==	Equivalence	Returns a True if both the left and right-hand operands are the same.
2	!=	Non-Equivalence	Returns True only if the left and right-hand operands are different.
1	>	Greater Than	Returns a True if the left-hand operand is larger than the right.
1	<	Less Than	Returns a True if the left-hand operand is smaller than the right.
1	>=	Greater Than or Equal	Returns a True if the left-hand operand is larger than or equal to the right.
1	<=	Less Than or Equal	Returns a True if the left-hand operand is smaller than or equal to the right.

Combining Comparisons and Negation

```
1 == 2 and 3 == 2 # False
1 != 2 and 3 == 2 # False
1 != 2 and 3 != 2 # True
1 == 1 and 2 == 2 # True
1 == 2 or 3 == 2 # False
1 != 2 or 3 == 2 # True
1 == 2 or 3 != 2 # True
1 != 2 or 3 != 2 # True
```

```
not 1 == 2 # True
not 1 != 2 # False

not 1 > 2 # True
not 1 < 2 # False

# Another place this is useful is for checking
# to see if an element is in a list
list1 = [1, 3, 5, 7, 11]
2 in list1 # False
3 in list1 # True
9 not in list1 # True
```

- It's useful to chain these comparisons together to form complex expressions.
- This can be done using the **and** and **or** operators.
- Remember, the not operator inverts their meaning; something that evaluates to True becomes False, and vice versa.

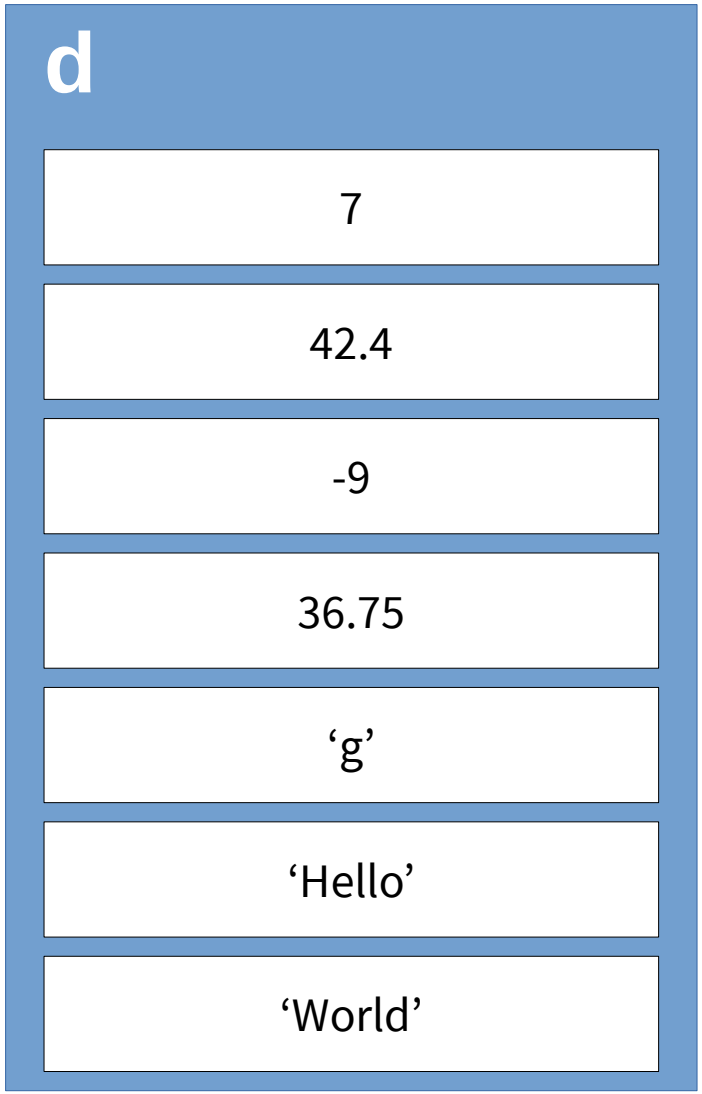
Types and Variables

- When we perform assignments, the value gets assigned to a variable; a container to hold things.
- Each variable has a type, depending on the value it’s holding.
- Python has useful built in, primitive types.
- Python is dynamically typed, meaning you don’t have to worry too much about a variables type; Python mostly deals with it for you.

Types

Name	Converters	Description
Boolean	bool()	A **True** or **False** value.
Integer	int()	Can contain any real, **whole** number.
Floating Point	float()	Can contain any real number.
String	str()	Can contain a string of Unicode (**UTF-8**) characters.
Complex	complex()	Contains a number with both a **real** and **imaginary** part. We won't be going into these, but know they exist.

Collections – Lists



Initialisation

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = [13.2, 95.35, 2634.27, -0.251, 0.00152]
c = ['Hello', 'world']
d = [7, 42.4, -9, 36.75, 'g', 'Hello', 'World']
```

Element Access / Mutation

```
a[5] = 23
b[3] # Outputs: -0.251
```

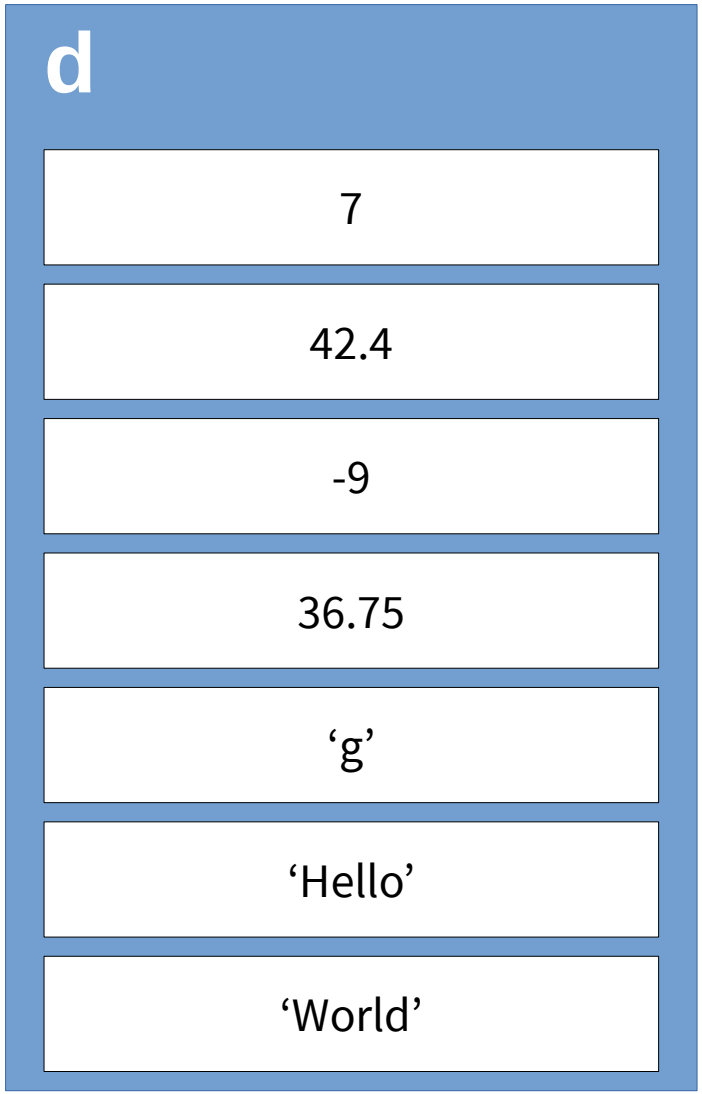
Appending

```
a.append(10) # Now a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Getting the Length

```
len([0, 1, 2, 3, 4]) # In this case, len() gives you 5.
```

Collections – Tuples



Initialisation

```
a = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
b = (13.2, 95.35, 2634.27, -0.251, 0.00152)
c = ('Hello', 'world')
d = (7, 42.4, -9, 36.75, 'g', 'Hello', 'World')
```

Access

```
b[1] # Gives 95.35
```

Getting the Length

```
len(b) # Gives 5
```

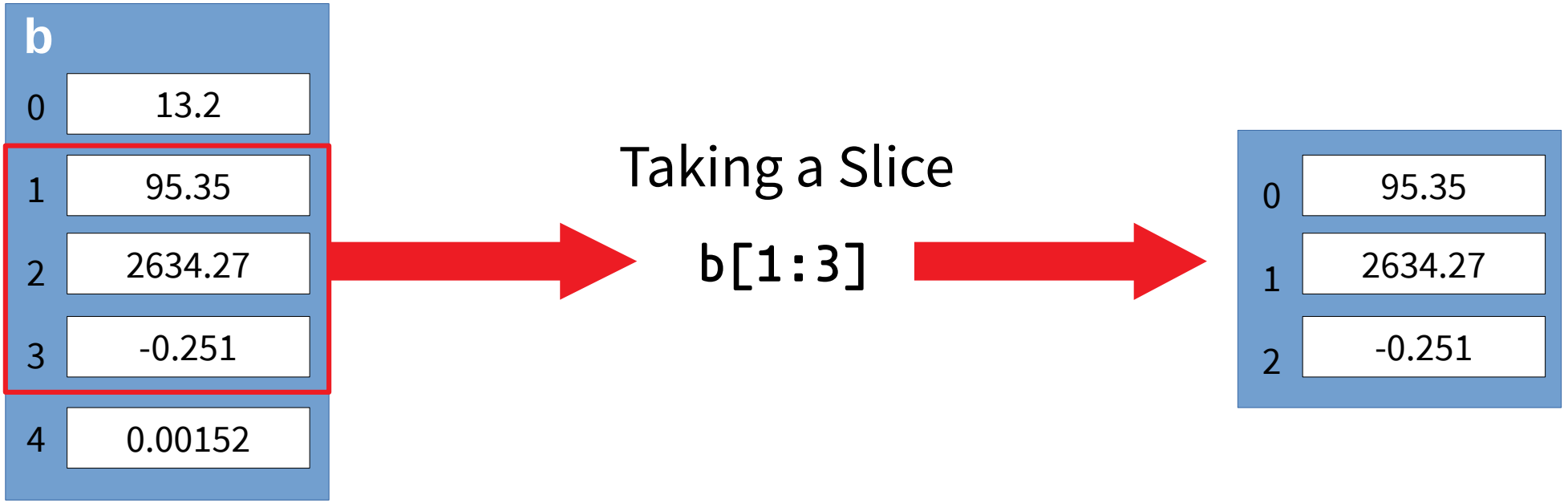
~~Mutation~~

```
b[1] = 42 # Causes an error!
b.append(42) # Causes an error!
```

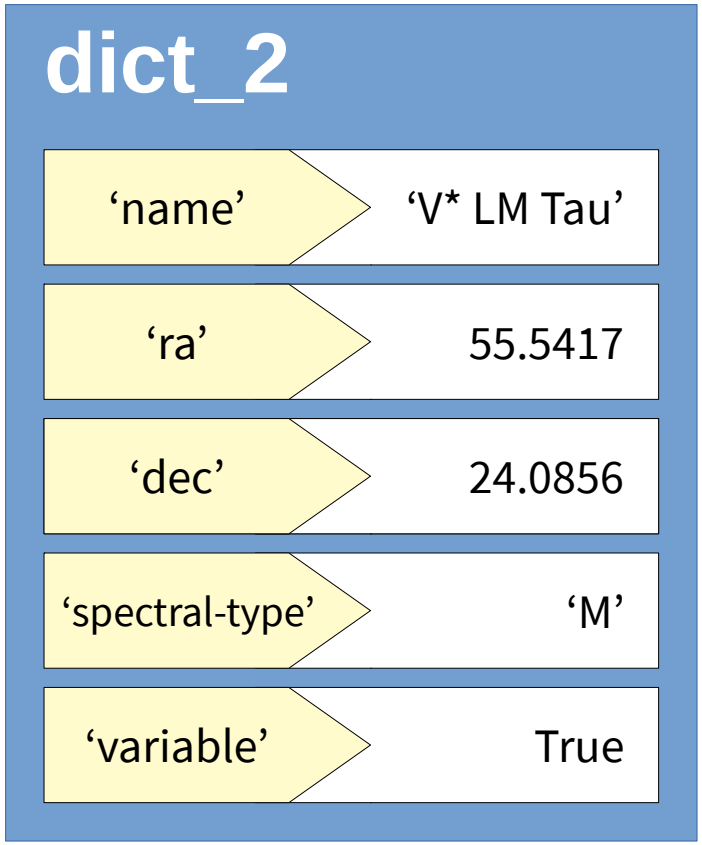
Collections - Slicing

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = (13.2, 95.35, 2634.27, -0.251, 0.00152)

# Now we can slice both of these collections to extract a subset of their elements.
a[1:6] # Gives [1, 2, 3, 4, 5, 6]
b[1:3] # Gives (95.35, 2634.27, -0.251)
```



Collections – Dictionaries



Initialisation

```
dict_2 = {
        'name' : 'V* LM Tau',
        'ra' : 55.5417,
        'dec' : 24.0856,
        'spectral-type' : 'M',
        'variable' : True
}
```

Element Access / Mutation

```
# Indiviual elements within dictionaries can be access like so
star_name = dict_2['name']
variable_star = dict_2['variable']

# And they can be set like so
dict_2['variable'] = False
dict_2['name'] = 'LM Tau'
```

- Note: Dictionaries do not store elements in order.

Combining Collections

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

combined_list = [list1, list2, list2]

print(combined_list[1]) # Will print [4, 5, 6]
print(combined_list[1][1]) # Will print 5
```

- You can actually nest containers within one another as well!

```
tuple2 = (4, 5, 6)
tuple3 = (7, 8, 9)
tuple1 = (1, 2, 3)

dictionary = { 'tuple1' : tuple1, : 'tuple2' : tuple2, 'tuple3' : tuple3 }

print(dictionary['tuple2']) # Gives (4, 5, 6)
print(dictionary['tuple2'][1]) # Gives 5
```

Control Flow

If Statements

```
if <expression>:
        <code block>
elif <expression>: # Optional
        <code block>
else: # Optional
        <code block>
```

If statements let you make decisions within your code. **Expressions** are tested until one returns true, then it's code block is executed.

- **The elif** and **else** clauses are optional.
- You can have as many **elif**s as you want.
- Remember to indent your code blocks.

If Statements - Example

```
# In this example we have an unknown integer and the user to guess what it is.
# We're going to make it easier and tell them whether it's higher or lower.
number = 42

guess = int(input('Guess the integer: '))

if number == guess:
        # Remember to indent, because this is a block controlled by the if statement.
        print("Well done, you got the number!")
elif number > guess:
        print('Sorry, your guess was incorrect. The number is higher.')
else:
        print('Sorry, your guess was incorrect. The number is lower.')
```

For Statements

```
for <variable> in <collection>:
        <code block>
```

For statements let you iterate over the elements of a **List** or **Tuple**.

- For each element in the list or tuple, the for loop will put the value into the variable and execute the code block with this variable.
- Remember, indent the code block.

For Statements - Example

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
square = []

for number in list:
        square.append(number ** 2.0)

print(square) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

While Statements

```
while <expression>:
        <code block>
```

While statements let you iterate a block of code for as long as an expression evaluates are true.

- This works well if you want to loop a block of code without iterating over the elements of an array.
- Remember, indent the code block.

While Statements - Example

```
# In this example we have an unknown integer and the user to guess what it is.
# We're going to make it easier and tell them whether it's higher or lower.
number = 42
guess = 0

while number != guess:
        guess = int(input('Guess the integer: '))

        if number == guess:
                print("Well done, you got the number!")
        elif number > guess:
                print('Sorry, your guess was incorrect. The number is higher. Try again.')
        else:
                print('Sorry, your guess was incorrect. The number is lower. Try again.')
```

Basic I/O

Print and Input

```
print("<text>")
```

The print function is your friend. You can use it to output messages and any values within your code.

```
input(<message>)
```

The input function allows you to read an input from the console. You can even give it a message to the user as an argument.

Print - Example

```
# Here are some examples of things that
# can be easily output using the print function in Python.
print("Just a string of text") # Just a string of text
print(42) # 42
print(24.5) # 24.5

boolean_value = True
print(boolean_value) # True

# You can even perform a calculation inside a print
print(17.4 + 34.1 - 10) # 41.5
```

Input - Example

```
# You can get a value from the console like this
name = input('What is your name?: ') # What is your name?: Sam

# You can chain this together with .format() and type
# conversions to make your program really interactive
age = int(input('Hello, {}. How old are you?: '.format(name)))

# Inputting a true or false here will convert it to a bool
member = bool(input('Are you a member of Space Exe, {}?: '.format(name))) # True
```

Concatenation and Format

You can add multiple strings and values together by adding them, this is called concatenation. This resulting string can then be conveniently printed.

```
string = 'Spa' + 'ce ' + 'Exe ' + 'is awesome!'
print(string) # Space Exe is awesome!
```

The more Pythonic way of putting values into your strings is to use string.format(). You can feed curly braces into your strings and feed the values in as arguments in format.

```
'{} {}...{}'.format(<value1>, <value2>,..., <valueN>)
```

Format - Example

```
string = '{} {}'.format('one', 'two')
print(string) # one two

print('Acceleration due to gravity: {} m/s'.format(9.81))

meaning_of_universe = 42
print('Meaning of the Universe: {}'.format(meaning_of_universe))
```

Reading and Writing a File

```
f = open(<filename>, <mode>)
<code to read the file>
f.close()
```

- First, we open a file using open(). The arguments are the path to the file, and the open mode.
- This returns a file object if successful, or **None** if not.
- When you're done with the file. Call the file objects **close()** method.

Mode	Meaning
r	read
w	write
x	exclusive creation
a	append
t	text mode
b	binary mode
+	updating

Reading a File - Example

```
filename = 'xydata.dat'
x = []
y = []

# First we open the file
f = open(filename, 'r')

# We can read the entire contents of the file into
# a string, like so
entire_file = f.read()

# Or we can read each line into an element in a list.
# Before that, we seek the start of the file to read it in again.
f.seek(0, 0)
rows_of_file = f.readlines()

# We can loop over the lines in a file like so
for row in rows_of_file:
        tmp = row.replace('\n', '')
        x_tmp, y_tmp = tmp.split(" ")
        x.append(float(x_tmp))
        y.append(float(y_tmp))

# Finally we close the file
f.close()

print(x) # [12.0, 23.0, 13.0, 12.0, 16.0]
print(y) # [562.0, 762.0, 87.0, 97.0, 212.0]
```

Writing a File - Example

```
# Code to write x and y coordinates.
x = [12, 23, 13, 12, 16]
y = [562, 762, 87, 97, 212]

# Opens the file for writing, creating it
# if it doesn't exist.
f = open('xydata.dat', 'w')

# Check that the file is open before we try to write
if f:
        # Iterate from 0 to length of the arrays minus 1
        # because the arrays are zero indexed.
        for i in range(0, len(x) - 1):
                f.write("{}\t{}\n".format(x[i], y[i]))

        # When we're done writing, we can close the file
        f.close()
```

Functions and Modules

Functions

```
def function_name(arg1, arg2):
        # contents of function go here
        print(arg1, ', ', arg2)

# You can call your function like this
function_name('Hello', 'World') # Hello, World
```

Functions let you bundle a chunk of code into a black and call it when you need it.

- You can provide the code with variables in the form of arguments, which are fed in between brackets.
- This means you can easily reuse your code instead of having to write it over and over again.

Returning from Functions

- You can also return values from functions with the **return** statement.

```
a = 0
def meaniing_of_universe():
        return 42

def area_of_cricle(radius):
        area = 3.14169265 * ( radius ** 2.0 )
        return area

print(meaning_of_universe()) # 42
a = area_of_cricle(2.5)
print(a) # 19.63
```

Modules

Without Modules

```
#script.py

def function1():
    return 42

def function2(r):
    area = pi * r ** 2.0
    return area

other_code()
f = open('file.txt')
str = f.read()
print(str)
function2(3.0)
```

With Modules

```
#module.py

def function1():
    return 42

def function2(r):
    area = pi * r ** 2.0
    return area
```

```
#script.py
import module

other_code()
f = open('file.txt')
str = f.read()
print(str)
module.function2(3.0)
```

Modules – The Module

```
# Circle module - circle.py

# Calculates and returns the area of a circle with radius r
def area(r):
        import math
        return math.pi * ( r ** 2.0 )

# Calculates and returns the circumference of a circle with radius r
def circumference(r):
        from math import pi
        return 2.0 * pi * r
```

Modules – Using the Module

Import Syntax

```
from <module> import <function / value>
```

To Import and use our **circle** module

```
from circle import area, circumference

r = 3.2
a = area(3.2)
print('Area: ', a) # Area: 32.17
c = circumference(r)
print('Circumference: ', c) # Circumference: 20.11
```

To Import all Functions from a Module

```
from circle import *
```

Writing Pythonic Code

Inline If

If you want to make a decision in a single line, such as in an assignment, Python lets you do an **inline if**.

```
<expression-if-true> if <condition> else <expression-if-false>
```

Example

```
# We can make simple decisions and assignments much easier using this construct
a = 10
print('Hello, World') if a == 10 else None # Hello, World
c = True if a > 5 else False # a = True

# You can combine it with the 'in' to see if something is in an array or dictionary.
committee  = ['Ben', 'Marine', 'Mark', 'Matt', 'Sam']
name = 'Jim'
print('On committee') if name in committee else print('Not on committee')
# Not on committee
```

List Comprehensions

If you want to quickly make a new list from an iterable object, such as a list, you can use **list comprehensions**.

```
<list> = [ <expression> for <variable> in <array / generator> if <conditional> ]
```

Example

```
# We can generate a new array using a generator, like the range function
array1 = [ x + 1 for x in range(10)  ] # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# We can generate a new list from the values of a previous list
array2 = [ i * 10 for i in array1 ] # [10, 20, 30, 40, 50, 60 70, 80, 90, 100]

# We can add in the conditional clause to filter off certain values
array3 = [ x for  x in array1 if x % 2 == 0 ] # [2, 4, 6, 8, 10]
```

Unpacking Return Values

```
def pedantic_square_root(number):
        from math import sqrt
        root = sqrt(number)
        return (root, -root)

# We can give only one variable to assigne and
# we just get the tuple returned
root = pedantic_square_root(4)
print('{}'.format(root)) # (2.0, -2.0)

# However, because the function returns a tuple
# we can unpack them into two variables
root1, root2 = pedantic_square_root(4)
print('{} {}'.format(root1, root2)) # 2.0 -2.0
```

You can easily get multiple return values from a function by packaging them in a **tuple** and **unpacking** them outside.

Zipping

You can easily iterate over more than one array using the zip() function. This makes outputting **x** and **y** values ,as with the writing example, a lot simpler.

```
# Zip combines the two or more arrys into a single
# for loop, leting you easily iterate over the values

list1 = ['a1', 'a2', 'a3', 'a4', 'a5']
list2 = ['b1', 'b2', 'b3', 'b4', 'b5']

for l1, l2 in zip(list1, list2):
        print('{}, {}'.format(l1, l2))
```

Enumeration

If you have too many arrays to **zip()**, **enumerate()** gives you the elements of a single list, as well as the current index, for conveniently accessing other, associated lists.

```
# Zip combines the two or more arrys into a single
# for loop, leting you easily iterate over the values

list1 = ['a1', 'a2', 'a3', 'a4', 'a5']
list2 = ['b1', 'b2', 'b3', 'b4', 'b5']

for i, l1 in enumerate(list1):
        l2 = list2[i]
        print('{}, {}'.format(l1, l2))
```

Python 2 vs. Python 3

Python 2 vs. Python 3 - Printing

```
#Python 2
print 'This is a Python 2 print' # Will throw an error in Python 3
# Python 3
print('This is a Python 3 print') # Will work in Python 2
```

- To improve flexibility, the print statement has been converted to a function in Python 3.
- A Python 2 style print will throw an error in Python 3, however a Python 3 style print will work in recent versions of Python 2.

Python 2 vs. Python 3 - Integer Division

```
# In Python 2:
print 1 / 2 # gives 0

# However, we can get around it using a type conversion:
print 1 / float(2) # gives 0.5

# Also, if we make one of the operands a float, it will
# output a float
print 1 / 2.0 # gives 0.5
```

- **In Python 2, integer division will result in an integer.**
- **We can either give convert implicitly to float, convert using float().**

```
from __future__ import division
# In Python 2:
print 1 / 2 # now gives 0.5
```

- ...or import division from __future__ to override the '*/*' operator.