Micropython Quick Start

1. What is Micropython

MicroPython is a streamlined and efficient implementation of the programming language Python3. Its syntax is consistent with Python3, but it only implements a small part of the Python standard library. MicroPython is optimized for use in the Internet of Things (IoT), consumer electronics, and embedded systems. Python is a weakly typed language with efficient and concise syntax. Before we start using our K230, let's first master some basic knowledge of Python syntax.

2. Python 3 Basics

This article is based on Louie Dinh's "Getting Started with Python 3 in 10 Minutes", translated by Geoff Liu

Notes

```
# Line comments start with a pound sign

""" Use three quotes for multi-line strings
    Packages are also often used to make multiple
    Line Comments
"""
```

Primitive data types and operators

```
####################################################
## 1. Primitive data types and operators
####################################################

# Integer
3   # => 3

# Nothing unexpected in arithmetic
1  +  1   # => 2
```

```
8  -  1   # => 7
10  *  2   # => 20

# But the division exception will be automatically converted into floating point numbers
35  /  5   # => 7.0
5  /  3   # => 1.6666666666666667

# The result of integer division is always rounded down
5  //  3      # => 1
5.0  //  3.0  # => 1.0 # Floating point numbers are also acceptable
- 5  //  3   # => -2
-5.0 //  3.0  #  => -2.0

# The result of floating-point operations is also a floating-point number
3  *  2.0  # => 6.0

# Modulo
7  %  3  # => 1

# x to the power of y
2 ** 4  # => 16

# Using parentheses to determine precedence
( 1  +  3 ) *  2   # => 8

# Boolean values
True
False

# Use not to negate
not  True   # => False
not  False   # => True

# Logical operators, note that and and or are lowercase
True  and  False  # => False
False  or  True  # => True

# Integers can also be treated as Boolean values
0  and  2  # => 0
- 5  or  0  # => -5
0 == False  # => True
2 == True  # => False
1 == True  # => True

# Use == to check equality
1 == 1   # => True
2 == 1   # => False

# Use != to judge inequality
1  ! = 1   # => False
2  ! = 1   # => True

# Compare sizes
1  <  10   # => True
1  >  10   # => False
2  < = 2   # => True
2  > = 2   # => True
```

```
# Size comparisons can be linked together!
1 < 2 < 3 # => True
2 < 3 < 2 # => False

# Strings can be quoted in single or double quotes
"This is a string"
'This is also a string'

# Concatenate strings using the plus sign
"Hello " + "world!" # => "Hello world!"

# Strings can be treated as lists of characters
"This is a string" [ 0 ] # => 'T'

# Use .format to format strings
"{} can be {}" . format ( "strings" , "interpolated" )

# Parameters can be repeated to save time
"{0} be nimble, {0} be quick, {0} jump over the {1}" . format ( "Jack" , "candle
stick" )
# => "Jack be nimble, Jack be quick, Jack jump over the candle stick"

# If you don't want to count parameters, you can use keywords
"{name} wants to eat {food}" . format ( name = "Bob" , food = "lasagna" )
# => "Bob wants to eat lasagna"

# If your Python3 program also needs to run in an environment below Python 2.5,
you can also use the old formatting syntax
"%s can be %s the %s way" % ( "strings" , "interpolated" , "old" )

# None is an object
None # => None

# When comparing with None, do not use ==, use is. is is used to compare whether
two variables point to the same object.
"etc" is None # => False
None is None # => True

# None, 0, empty string, empty list, empty dictionary are all considered False
# All other values are True
bool ( 0 ) # => False
bool ( "" ) # => False
bool ([]) # => False
bool ({}) # => False
```

Variables and Collections

```
####################################################
## 2. Variables and collections
####################################################

# print is a built-in printing function
print ( "I'm Python. Nice to meet you!" )
```

```
# No need to declare variables before assigning them values
# Traditional variable names are lowercase, with underscores separating words
some_var = 5
some_var   # => 5

# Accessing an unassigned variable will throw an exception
# Refer to the process control section to learn exception handling
some_unknown_var   # throws a NameError

# Using lists to store sequences
li = []
# You can also assign elements when creating a list
other_li = [ 4 , 5 , 6 ]

# Use append to add elements to the end of the list
li . append ( 1 )     # li is now [1]
li . append ( 2 )     # li is now [1, 2]
li . append ( 4 )     # li is now [1, 2, 4]
li . append ( 3 )     # li is now [1, 2, 4, 3]
# Use pop to remove from the end of the list
li . pop ()         # => 3 and li is now [1, 2, 4]
# Put 3 back
li . append ( 3 )     # li changes back to [1, 2, 4, 3]

# List access is the same as array access
li [ 0 ]   # => 1
# Take out the last element
li [ - 1 ]   # => 3

# Out-of-bounds access will cause IndexError
li [ 4 ]   # throws IndexError

# Lists have a slice syntax
li [ 1 : 3 ]   # => [2, 4]
# Take the tail
li [ 2 :]   # => [4, 3]
# Get the header
li [: 3 ]   # => [1, 2, 4]
# Take every other one
li [:: 2 ]   # =>[1, 4]
# Inverted list
li [:: - 1 ]   # => [3, 4, 2, 1]
# You can use any combination of the three parameters to construct a cut
# li[beginning:end:step]

# Use del to delete any element
del  li [ 2 ]    # li is now [1, 2, 3]

# Lists can be added
# Note: The values of li and other_li remain unchanged
li  +  other_li    # => [1, 2, 3, 4, 5, 6]

# Use extend to concatenate lists
li . extend ( other_li )    # li is now [1, 2, 3, 4, 5, 6]

# Use in to test if a list contains a value
1  in  li    # => True
```

```
# Use len to get the length of the list
len ( li )    # => 6


# Tuples are immutable sequences
tup = ( 1 , 2 , 3 )
tup [ 0 ]    # => 1
tup [ 0 ] = 3   # throws TypeError

# Lists allow most operations on tuples.
len ( tup )    # => 3
tup  + ( 4 , 5 , 6 )    # => (1, 2, 3, 4, 5, 6)
tup [: 2 ]    # => (1, 2)
2  in  tup    # => True

# You can unpack the tuple list and assign it to a variable
a , b , c = ( 1 , 2 , 3 )      # Now a is 1, b is 2, c is 3
# The parentheses around tuples are optional
d , e , f = 4 , 5 , 6
# Swapping the values of two variables is as simple as that
e , d = d , e      # Now d is 5, e is 4


# Use dictionary to express mapping relationship
empty_dict = {}
# Initialized dictionary
filled_dict = { "one" : 1 , "two" : 2 , "three" : 3 }

# Use [] to get the value
filled_dict [ "one" ]    # => 1


# Use keys to get all keys.
# Because keys returns an iterable object, we wrap the result in a list here. We
will introduce iterables in detail below.
# Note: The order of dictionary keys is not fixed, and your results may be
different from the following.
list ( filled_dict . keys ())    # => ["three", "two", "one"]


# Use values to get all the values. Like keys, they need to be wrapped in a list
and the order may be different.
list ( filled_dict . values ())    # => [3, 2, 1]


# Use in to test whether a dictionary contains a key
"one"  in  filled_dict    # => True
1  in  filled_dict    # => False

# Accessing a non-existent key will result in a KeyError
filled_dict [ "four" ]    # KeyError

# Using get to avoid KeyError
filled_dict . get ( "one" )    # => 1
filled_dict . get ( "four" )    # => None
# when the key does not exist, the get method can return a default value
filled_dict . get ( "one" , 4 )    # => 1
```

```
filled_dict . get ( "four" , 4 )    # => 4

# The setdefault method only inserts a new value when the key does not exist
filled_dict . setdefault ( "five" , 5 )   # filled_dict["five"] is set to 5
filled_dict . setdefault ( "five" , 6 )   # filled_dict["five"] is still 5

# Dictionary assignment
filled_dict . update ({ "four" : 4 }) # => {"one": 1, "two": 2, "three": 3,
"four": 4}
filled_dict [ "four" ] = 4   # Another way to assign values

# Delete with del
del  filled_dict [ "one" ]   # delete one from filled_dict


# Using set to express collections
empty_set = set ()
# Initialize a set, the syntax is similar to that of a dictionary.
some_set = { 1 , 1 , 2 , 2 , 3 , 4 }    # some_set is now {1, 2, 3, 4}

# You can assign a collection to a variable
filled_set = some_set

# Add elements to the collection
filled_set . add ( 5 )    # filled_set is now {1, 2, 3, 4, 5}

# & Get the intersection
other_set = { 3 , 4 , 5 , 6 }
filled_set  &  other_set    # => {3, 4, 5}

# | Take the union
filled_set  |  other_set    # => {1, 2, 3, 4, 5, 6}

# - Take the complement of a set
{ 1 , 2 , 3 , 4 } - { 2 , 3 , 5 }    # => {1, 4}

# in tests whether a collection contains an element
2  in  filled_set    # => True
10  in  filled_set    # => False
```

Flow Control and Iterators

```
####################################################
## 3. Flow Control and Iterators
####################################################

# First define a variable
some_var = 5

# This is an if statement. Note that indentation is meaningful in Python.
# Print "some_var is less than 10"
if  some_var  >  10 :
    print ( "some_var is greater than 10" )
elif  some_var  <  10 :     # the elif clause is optional
    print ( "some_var is less than 10" )
else :                      # else is also optional
```

```
    print ( "some_var is 10" )


"""
Iterating over a list using a for loop
Print:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for  animal  in [ "dog" , "cat" , "mouse" ]:
    print ( "{} is a mammal" . format ( animal ))

"""
"range(number)" returns a list of numbers from 0 to the given number
Print:
    0
    1
    2
    3
"""
for  i  in  range ( 4 ):
    print ( i )

"""
while loop until the condition is not met
Print:
    0
    1
    2
    3
"""
x = 0
while  x  <  4 :
    print ( x )
    x += 1   # Shorthand for x = x + 1

# Handle exceptions with a try/except block
try :
    # Throwing an exception with raise
    raise  IndexError ( "This is an index error" )
except  IndexError  as  e :
    pass     # pass is a no-op, but errors should be handled here
except ( TypeError , NameError ):
    pass     # can handle different types of errors at the same time
else :    #else statement is optional and must be after all except
    print ( "All good!" )    # This statement will only be executed if try
completes without error


# Python provides a basic abstraction called iterable. An iterable object can be
treated as a sequence
# Object. For example, the object returned by range above is iterable.

filled_dict = { "one" : 1 , "two" : 2 , "three" : 3 }
our_iterable = filled_dict . keys ()
print ( our_iterable ) # => dict_keys(['one', 'two', 'three']), an object that
implements the iterable interface
```

```
# Iterable objects can be traversed
for  i  in  our_iterable :
    print ( i )     # print one, two, three

# But random access is not possible
our_iterable [ 1 ]   # throws TypeError

# Iterable objects know how to generate iterators
our_iterator = iter ( our_iterable )

# An iterator is an object that remembers the position of the traversal
# Use __next__ to get the next element
our_iterator . __next__ ()   # => "one"

# The position will be remembered when calling __next__ again
our_iterator . __next__ ()   # => "two"
our_iterator . __next__ ()   # => "three"

# When all elements of the iterator are taken out, StopIteration will be thrown
our_iterator . __next__ () # throws StopIteration

# You can use list to retrieve all elements of the iterator at once
list ( filled_dict . keys ())   # => Returns ["one", "two", "three"]
```

function

```
####################################################
## 4. Functions
####################################################

# Define new functions with def
def  add ( x , y ):
    print ( "x is {} and y is {}" . format ( x , y ))
    return  x  +  y     # return with return statement

# Calling the function
add ( 5 , 6 )    # => prints "x is 5 and y is 6" and returns 11

# You can also call functions with keyword arguments
add ( y = 6 , x = 5 )    # keyword arguments can be used in any order


# We can define a variable parameter function
def  varargs ( * args ):
    return  args

varargs ( 1 , 2 , 3 )    # => (1, 2, 3)


# We can also define a keyword variable parameter function
def  keyword_args ( ** kwargs ):
    return  kwargs
```

```
# Let's see what the result is:
keyword_args ( big = "foot" , loch = "ness" )    # => {"big": "foot", "loch":
"ness"}


# These two variable parameters can be mixed
def  all_the_args ( * args , ** kwargs ):
    print ( args )
    print ( kwargs )
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# When calling a variable parameter function, you can do the opposite of the
above, using * to expand the sequence and ** to expand the dictionary.
args = ( 1 , 2 , 3 , 4 )
kwargs = { "a" : 3 , "b" : 4 }
all_the_args ( * args )    # equivalent to foo(1, 2, 3, 4)
all_the_args ( ** kwargs )    # equivalent to foo(a=3, b=4)
all_the_args ( * args , ** kwargs )    # equivalent to foo(1, 2, 3, 4, a=3, b=4)


# Function Scope
x = 5

def  setX ( num ):
    # The local scope x is different from the global scope x
    x = num  # => 43
    print ( x ) # => 43

def  setGlobalX ( num ):
    global  x
    print ( x ) # => 5
    x = num  # Now the global x is assigned
    print ( x ) # => 6

setX ( 43 )
setGlobalX ( 6 )

# Functions are first-class citizens in Python
def  create_adder ( x ):
    def  adder ( y ):
        return  x  +  y
    return  adder

add_10 = create_adder ( 10 )
add_10 ( 3 )    # => 13

# There are also anonymous functions
( lambda  x : x  >  2 )( 3 )    # => True

# Built-in higher-order functions
map ( add_10 , [ 1 , 2 , 3 ])    # => [11, 12, 13]
filter ( lambda  x : x  >  5 , [ 3 , 4 , 5 , 6 , 7 ])    # => [6, 7]
```

```
# Using list comprehensions can simplify mapping and filtering. The return value
of a list comprehension is another list.
[ add_10 ( i ) for  i  in [ 1 , 2 , 3 ]]   # => [11, 12, 13]
[ x  for  x  in [ 3 , 4 , 5 , 6 , 7 ] if  x  >  5 ]    # => [6, 7]
```

Modules

Note that MicroPython cannot use pip to download third-party modules.

```
####################################################
## 5. Modules
####################################################

# Importing modules with import
import  math
print ( math . sqrt ( 16 ))   # => 4.0

# You can also import individual values from modules
from  math  import  ceil , floor
print ( ceil ( 3.7 ))   # => 4.0
print ( floor ( 3.7 ))    # => 3.0

# You can import all values in a module
# Warning: This is not recommended
from  math  import  *

# Abbreviate module names like this
import  math  as  m
math . sqrt ( 16 ) == m . sqrt ( 16 )    # => True

# Python modules are just regular Python files. You can write them yourself and
then import them.
# The name of the module is the name of the file.

# You can list all the values in a module like this
import  math
dir ( math )
```

kind

```
####################################################
## 6. Classes
####################################################


# Define a class that inherits object
class  Human ( object ):
```

```
    # Class attributes, shared by all instances of this class.
    species = "H. sapiens"

    # Constructor, called when the instance is initialized. Note the double
underscores before and after the name, which indicates that this property
    # Properties or methods have special meanings to Python, but are allowed to
be defined by the user. You should not use these when naming your own
    # formats.
    def  __init__ ( self , name ):
        # Assign the argument to the instance's name attribute
        self . name = name

    # Instance method, the first parameter is always self, which is the instance
object
    def  say ( self , msg ):
        return  "{name}: {message}" . format ( name = self . name , message =
msg )

    # Class method, shared by all instances of this class. The first parameter is
the class object.
    @classmethod
    def  get_species ( cls ):
        return  cls . species

    # Static method. There is no instance or class binding when calling.
    @staticmethod
    def  grunt ():
        return  "*grunt*"


# Construct an instance
i = Human ( name = "Yahboom" )
print ( i . say ( "hi" ))      # Prints "Yahboom: hi"

j = Human ( "xzt" )
print ( j . say ( "hello" ))   # Prints "xzt: hello"

# Calling a class method
i . get_species ()    # => "H. sapiens"

#Change a shared class attribute
Human . species = "H. neanderthalensis"
i . get_species ()    # => "H. neanderthalensis"
j . get_species ()    # => "H. neanderthalensis"

# Calling static methods
Human . grunt ()    # => "*grunt*"
```

Class inheritance

```
# Define a class that inherits object
class  Human ( object ):

    # Class attributes, shared by all instances of this class.
    species = "H. sapiens"
```

```
    # Constructor, called when the instance is initialized. Note the double
underscores before and after the name, which indicates that this property
    # Properties or methods have special meanings to Python, but are allowed to
be defined by the user. You should not use these when naming your own
    # formats.
    def  __init__ ( self , name ):
        # Assign the argument to the instance's name attribute
        self . name = name

    # Instance method, the first parameter is always self, which is the instance
object
    def  say ( self , msg ):
        return  "{name}: {message}" . format ( name = self . name , message =
msg )

    # Class method, shared by all instances of this class. The first parameter is
the class object.
    @classmethod
    def  get_species ( cls ):
        return  cls . species

    # Static method. There is no instance or class binding when calling.
    @staticmethod
    def  grunt ():
        return  "*grunt*"


# Construct an instance
i = Human ( name = "Yahboom" )
print ( i . say ( "hi" ))      # Prints "Yahboom: hi"

j = Human ( "xzt" )
print ( j . say ( "hello" ))   # Prints "xzt: hello"

# Calling a class method
i . get_species ()    # => "H. sapiens"

#Change a shared class attribute
Human . species = "H. neanderthalensis"
i . get_species ()    # => "H. neanderthalensis"
j . get_species ()    # => "H. neanderthalensis"

# Calling static methods
Human . grunt ()    # => "*grunt*"
# The inheritance mechanism allows subclasses to inherit methods and variables
from parent classes.
# We can use the Human class as a base class or a parent class.
# Then define a subclass named Superhero to inherit the properties of the parent
class such as "species", "name", and "age"
# and methods such as "sing" and "grunt", and can also define its own unique
properties

# Based on the modularity of Python files, you can put this class in a separate
file, for example, human.py.

# To import functions from other files, you need to use the following statement
# from "filename-without-extension" import "function-or-class"
```

```
#Specify the parent class as a parameter for class initialization
class  Superhero ( Human ):

    # If the subclass needs to inherit all parent class definitions and does not need to make any modifications,
    # You can use the "pass" keyword directly (and no other statements are needed)
    # But it will be commented out in this example to generate different subclasses.
    # pass

    # Subclasses can override fields defined by parent classes
    species = 'Superhuman'

    # A subclass will automatically inherit the parent class's constructor including its parameters, but at the same time, a subclass can also add additional parameters or definitions.
    # Even override parent class methods such as constructors.
    # This constructor inherits the "name" parameter from the parent class "Human", and also adds "superpower" and
    # "movie" parameters:
    def  __init__ ( self , name , movie = False ,
                 superpowers =[ "super strength" , "bulletproofing" ]):

        # Add additional class parameters
        self . fictional = True
        self . movie = movie
        # Be careful with mutable default values, since default values are shared
        self . superpowers = superpowers

        # The "super" function allows you to access methods in the parent class that are overridden by the child class
        # In this example, the __init__ method is overridden
        # This statement is used to run the parent class's constructor:
        super (). __init__ ( name )

    # Override the sing method in the parent class
    def  sing ( self ):
        return  'Dun, dun, DUN!'

    # Add an additional method
    def  boast ( self ):
        for  power  in  self . superpowers :
            print ( "I wield the power of {pow}!" . format ( pow = power ))


if  __name__ == '__main__' :
    sup = Superhero ( name = "Tick" )

    # Check instance type
    if  isinstance ( sup , Human ):
        print ( 'I am human' )
    if  type ( sup ) is  Superhero :
        print ( 'I am a superhero' )
```

```
# Call the parent class method and use the child class's attributes
print ( sup . get_species ())     # => Superhuman

# Calling the overridden method
print ( sup . sing ())            # => Dun, dun, DUN!

# Calling Human methods
sup . say ( 'Spoon' )             # => Tick: Spoon

# Calling Superhero's unique methods
sup . boast ()                  # => I wield the power of super strength!
                            # => I wield the power of bulletproofing!

# Inherited class fields
sup . age = 31
print ( sup . age )               # => 31

# Superhero-specific fields
print ( 'Am I Oscar eligible? '  +  str ( sup . movie ))
```

3. Micropython other related API usage

You can refer to the API manual provided by Canaan Technology (K230 chip R&D company):

API Manual — CanMV K230