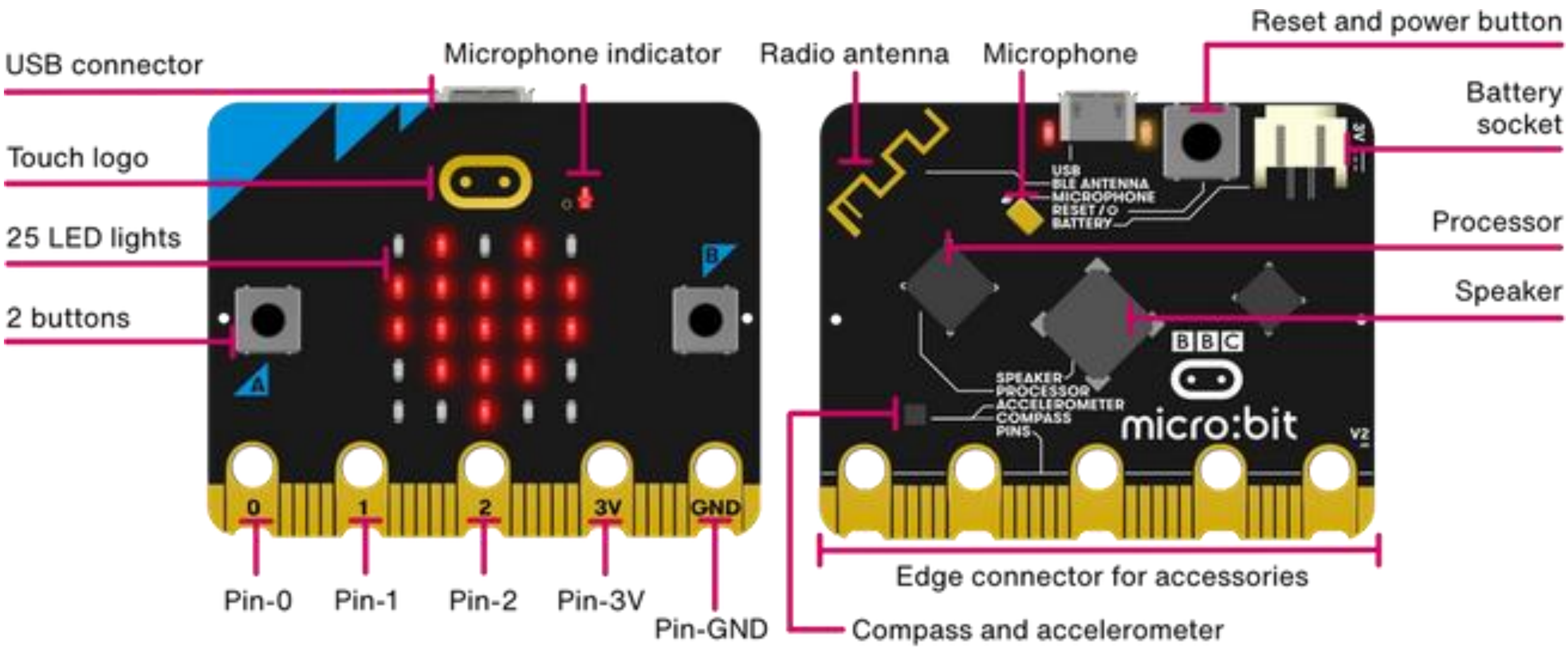
micro:bit TI-Nspire CX II & TI-84 CE Python - micro:bit Reference Guide

The built-in Python 3.x programming language supports a tethered connection to the V1 and V2 micro:bit boards. Functionality with the micro:bit card is facilitated by importing the microbit module(s) into a python program. The TI micro:bit module(s) syntax is aligned with the micro:bit's Python syntax posted on the micro:bit website. For more information and examples, please reference the website links below. Some methods are supported only on the TI Nspire CX II or only the TI-84 CE and some only on the micro:bit V2. Reference the BBC microbit MicroPython API for differences between V1 and V2.



BBC microbit MicroPython API – This is the official standard Python syntax for the micro:bit.
Micro:bit User Guide – The micro:bit.org website with lots of examples and tutorials.
Micro:bit Projects Make it: Code it– Projects ideas for use with the calculator. Python code is listed in Step 2 under the Python tab.
Texas Instruments Micro:bit Support Page with microbit module and runtime downloads and a Getting Started Guide.
Texas Instruments STEM Projects – Many engaging STEM educational activities.

API	Example	Notes
Commands		
from microbit import *	**from microbit import ***	Importing is required for all microbit functionality. After all micro:bit modules have been loaded onto the calculator, import the microbit module using the Add-On menu tab. All additional microbit support modules are added from the microbit module menu.
sleep(milliseconds)	**sleep(2000)**	The program execution will pause for the given number of milliseconds. Note the micro:bit sleep unit is mSec. **DO NOT** import the time module that uses the unit seconds.
while not escape():	**While not escape():** **x = accelerometer.get_x()** **print(x)**	**TI-84 Plus CE Python Only** - An indefinite loop control structure that exits when the calculator's [clear] key is pressed. This structure is an alternative to a while True loop.
while not get_key != 'esc':	**while not get_key != 'esc':** **x = accelerometer.get_x()** **print(x)**	**TI-Nspire CX II Only -** An indefinite loop control structure that exits when the calculator's [esc] key is pressed. This structure is an alternative to a while True loop.
disp_clr()	**disp_clr()**	**TI-84 Plus CE Python Only** - Clears the calculator's shell history.
clear_history()	**disp_clr()**	**TI-Nspire CX II Only** - Clears the calculator's shell history.
store_list("1" – "6", list)	**store_list("1",my_list)**	**TI-84 Plus CE Python Only** - Stores a Python list into the calculator's statistics list. The parameter "1" – "6" is used to select L1 through L6 on the CE. The "Name" parameter on the CX is and valid nspire name.
store_list("Name",list)	**store_list("nspire_name",my_list)**	**TI-Nspire CX II Only** - Stores a Python list into the calculator's statistics list. The parameter "1" – "6" is used to select L1 through L6 on the CE. The "Name" parameter on the CX is and valid nspire name.
var = temperature()	**t = temperature()**	Returns the micro:bit's CPU temperature in Celcius degrees.
Display		
Import from microbit module	**from mb_disp import ***	**TI-84 Plus CE Python Only** - Required micro:bit module for display methods.

.show(value)	**display.show(Image.HAPPY)**	Displays an image on the 5 x 5 display. An optional delay may be typed in that pauses the program for the specified time. For example delay = 500 pauses for 500 mSec. **`display.show(Image.HAPPY, DELAY = 500)`**
.scroll(value)	**display.scroll("Hello World!")**	Scrolls a number or a string across the display.
.clear()	**display.clear()**	Clears the 5 x 5 LED display.
.set_pixel(x,y,value)	**display.set_pixel(2,2,9)**	Turns on a pixel at location x,y (0-4) with the desired intensity (0-9). The center pixel is at (2,2).
var =.read_light_level()	**I = display.read_light_level()**	intensity = display.read_light_level()
var=Image(':':':':':')	**img=Image('99999:' '77777:' '55555:' '33333:' '11111:')**	Constructs a named image object. Each pixel in rows 1-5 must be set to brightness from 0 to 9. To enter values, position the cursor in front of the colon and enter the five values. Then reposition to the next row and complete for all five rows. This image can be displayed with display.show(img).
Image	**"Image.HEART"**	Pastes built-in image names. To display an image, use the display.show(value) method.
Music and Notes		
Import from microbit module	**from mb_music import * from mb_notes import ***	**TI-84 Plus CE Python Only** - Enables music and notes methods.
.play(value)	**music.play('music.ODE')**	Plays either a built-in song or a user-defined song. Use var=[note,] and then select notes from the Notes list to compose a song.
.pitch(frequency, duration)	**music.pitch(440,2000)**	Plays a tone of a given frequency and duration on mSec.
.set_tempo(ticks,BPM)	**music.temp(4,220)**	Changes the speed a song is played. Ticks are the base time unit, e.g., a quarter note is 4 ticks. BPM is the number of beats per minute; the typical tempo is 120 BPM.
.set_volume(0-255)	**music.set_volume (128)**	Sets the loudness of the built-in speaker. Silent is 0, and loudest is 255.
var=[note,]	**Frere_Jaques=['C4:4','D4:4','E4:4', 'C4:4','C4:4','D4:4','E4:4','C4:4', 'E4:4','F4:4','G4:8','E4:4','F4:4', 'G4:8',]**	A list of the first many notes of Frere Jaques. This list can be played using music.play(Frere_Jaques).

Songs	**'music.ODE'**	Pastes built-in song titles. To play a song, use the music.play(value) method. The 84 CE departs from the microbit API on song name format. The song name is pasted within quotes on the calculator; this may cause an error when copying programs from another source which may ommit the single quotes.
Notes	**'C4:4'**	**TI-84 Plus CE Python Only** - Pastes notes from octave 1 through octave 6 Pasting more than one note into the Music.play() method requires the notes to be within a list. See the var=[note,] helper above.
Audio		
Import from microbit module	**from mb_audio import ***	**TI-84 Plus CE Python Only** - Enables audio and sounds methods.
.play(sound)	**audio.play("Sound.GIGGLE")**	Plays either a built-in sound. Use the Sounds menu to select sound titles.
.stop()	**audio.stop()**	Halts a sound in progress
Sounds	**"Sound.GIGGLE"**	Pastes built-in sound titles. To play a sound, use the audio.play(value) method.
Microphone		
Import from microbit module	**from mb_mic import ***	**TI-84 Plus CE Python Only** - Enables microphone and sound events methods.
var=.sound_level()	**i=microphone.sound_level()**	Measures the present sound intensity in a range from 0 (quiet) to 255 (loud).
var=.current_event()	**event=microphone.current_event(SoundEvent.LOUD)**	Returns the named SoundEvent at the moment of execution.
.is_event(SoundEvent)	**microphone.is_event(SoundEvent.LOUD)**	Returns True if the microphone detects a sound pressure greater than SoundEvent.LOUD. The sound pressure of the SoundEvent is set using the microphone.set_threshold() method. This method polls the microphone when executed. To catch a fast transient event, use the microphone.was_event() method.

.was_event(SoundEvent,value)	**microphone.was_event(SoundEvent .LOUD)**	Returns True if the microphone detects a sound pressure greater than SoundEvent.LOUD. The sound pressure of the SoundEvent can be set using the microphone.set_threshold() method. This method runs in the background, and the event is cleared once the method is called. For immediate polling, use the microphone.is_event() method.
.set_threshold(SoundEvent,value)	**microphone.set_threshold(SoundEvent.LOUD, 128)**	Sets a SoundEvent threshold from 0 (silent) to 255 (loud) This is useful when using the microphone.current_event() or the microphone.is_event()
SoundEvent	**SoundEvent.LOUD / QUIET**	Pastes the event name into an editable method.
Buttons and Touch		
Import from microbit module	**from mb_butns import ***	**TI-84 Plus CE Python Only** - Enables buttons and touch methods.
button_a.is_pressed()	**while not button_a.is_pressed(): statements**	Returns True when the button is pressed; otherwise, False. This method is often used in place of a boolean conditional, where hardware is used to change program flow. The .is_pressed() method polls the button and returns the current state of the button.
button_a.was_pressed()	**if button_a.was_pressed(): statements**	Returns True when the button was pressed; otherwise, False. The .was_pressed() method runs in the background and is helpful to catch transient button presses or taps. The button state is immediately cleared after the method is called.
var=button_a.get_presses()	**presses = button_a.get_presses()**	Returns the number of button presses. Calling this method will clear the accumulator and reset the return value to zero.
button_b.is_pressed()	**while not button_b.is_pressed(): statements**	Identical action as button_a.
button_b.was_pressed()	**if button_b.was_pressed(): statements**	Identical action as button_a.
var=button_b.get_presses()	**presses = button_b.get_presses()**	Identical action as button_a.

pin_logo.is_touched()	**while pin_logo.is_touched():** **statements**	Returns True when the capacitive tough logo is touched; otherwise, False. This method is often used in place of a boolean conditional, where hardware changes program flow. The .is_touched() method immediately polls the logo and returns the current state of the sensor.
Sensors		
Import from microbit module	**from mb_sensr import ***	**TI-84 Plus CE Python Only** - Enables internal sensor methods.
var = accelerometer.get_x()	**x=accelerometer.get_x()**	Returns x-axis accelerometer measurement with a range of +/- 2000 mg.
var = accelerometer.get_y()	**y=accelerometer.get_y()**	Returns y-axis accelerometer measurement with a range of +/- 2000 mg.
var = accelerometer.get_z()	**z=accelerometer.get_z()**	Returns z-axis accelerometer measurement with a range of +/- 2000 mg.
var_x,var_y,var_z = accelerometer.get_values()	**x,y,z=accelerometer.get_values()**	Concurrently returns all three accelerometer axes measurements.
var = accelerometer.magnitude()	**mag=accelerometer.magnitude()**	Returns the sum of the normalized magnitudes of all three accelerometer axes. This method helps detect three-dimensional motion, such as a shake.
var = compass.heading()	**direction=compass.heading()**	Returns the compass heading from 0-360 degrees.
var = compass.get_x()	**X=compass.get_x()**	Returns the magnetometer's x-axis value in nano Tesla as a positive or negative integer, depending on the direction of the field.
var = compass.get_y()	**Y=compass.get_y()**	Returns the magnetometer's y-axis value in nano Tesla, as a positive or negative integer, depending on the direction of the field.
var = compass.get_z()	**Z=compass.get_z()**	Returns the magnetometer's z-axis value in nano Tesla, as a positive or negative integer, depending on the direction of the field.
var = compass.is_calibrated()	**is_cal=compass.is_calibrated()**	Returns True if the compass is calibrated, otherwise False.
var = compass.get_field_strength()		Returns the magnitude of the magnetic field around the card in nano Tesla

compass.calibrate()	**compass.calibrate()**	Forces a compass calibration of the card. After the method is processed, the card will scroll calibration instructions on the 5 x 5 LED display.
compass.clear_calibration()	**compass.clear_calibration()**	Deletes the stored compass calibration.
accelerometer.current_gesture()	**g=accelerometer.current_gesture()**	Returns the current gesture of 'up,' 'down,' 'left,' 'right,' 'face up,' 'face down, or 'shake.'
accelerometer.is_gesture()	**accelerometer.is_gesture('shake')**	Returns True if current gesture matches the argument gesture, otherwise False. The .is_gesture() polls immediately.
accelerometer.was_gesture()	**accelerometer.was_gesture('up')**	Returns True if the argument gesture matches the current gesture at any time since the method was last called, otherwise False.
Radio		
Import from microbit module	**from mb_radio import ***	**TI-84 Plus CE Python Only** - Enables internal radio methods.
.on()	**radio.on()**	Turns the 2.4 MHz radio on.
.off()	**radio.off()**	Turns the 2.4 MHz radio off.
.config(length,channel=7,power=6,group=0)	**radio.config(length=32,channel=7,power=6,group=0)**	Configures the radio modes. The length is the message character length. The channel is the radio's operational frequency, the transmit power, and the group is a packet routing setting. For two micro:bits to communicate, they must share both channel and group. The higher the power, the further the signal is broadcast.
.send("string")	**radio.send("Hello World!")**	Transmitts a string. Must be paired with radio.receive() on the other receiving calculator.
var =.receive()	**msg=radio.receive()**	Receives as string from the radio buffer. The buffer can hold multiple messages in a first in-first out configuration. If there is no message in the buffer, a None is returned.
.send_number (val)	**radio.send_number(3.1415)**	Transmitts a integer or floating point number. Must be paired with radio.receive_number() on the other receiving calculator.
var=radio.receive_number()	**number=radio.receive_number**	Receives an integer or floating point number. If there is no message in the buffer, a None is returned.

var=radio.receive_full()	**msg=radio.full()**	Returns a tuple containing three values, the message, RSSI, and timestamp, of the next entry in the buffer. If there are no pending messages then None is returned.
Input/Output Pins		
Import from microbit module	**from mb_pins import ***	**TI-84 Plus CE Python Only** - Enables analog and digital methods on I/O pins.
var = pin.read_digital()	**state_0 = pin0.read_digital()**	Returns the digital state of the pin0 as 0 (GND) or 1 (3.3V).
var = pin.read_analog()	**ADC_0 = pin0.read_analog()**	Returns the analog value of the voltage on pin0 as 0 (GND) to 1023 (3.3V).
pin.write_digital(value)	**pin0.write_digital(1)**	Sets the voltage on the digital output pin0 to GND (0) to 3.3V (1).
pin.write_analog(value)	**pin0.write_analog(128)**	Sets the voltage on the analog output pin0 to GND (0) to 3.3V (1023)
var = pin.read_digital()	**state_1 = pin1.read_digital()**	Returns the digital state of the pin1 as 0 (GND) or 1 (3.3V).
var = pin.read_analog()	**ADC_1 = pin1.read_analog()**	Returns the analog value of the voltage on pin1 as 0 (GND) to 1023 (3.3V).
pin.write_digital(value)	**pin1.write_digital(0)**	Sets the voltage on the digital output pin1 to GND (0) to 3.3V (1).
pin.write_analog(value)	**pin1.write_analog(511)**	Sets the voltage on the analog output pin1 to GND (0) to 3.3V (1023)
var = pin.read_digital()	**state_2= pin2.read_digital()**	Returns the digital state of the pin2 as 0 (GND) or 1 (3.3V).
var = pin.read_analog()	**ADC_2 = pin2.read_analog()**	Returns the analog value of the voltage on pin2 as 0 (GND) to 1023 (3.3V).
pin.write_digital(value)	**pin2.write_digital(1)**	Sets the voltage on the digital output pin2 to GND (0) to 3.3V (1).
pin.write_analog(value)	**pin2.write_analog(750)**	Sets the voltage on the analog output pin2 to GND (0) to 3.3V (1023)
pin.set_analog_period(value)	**pin1.set_analog_period(10)**	Sets the period of the PWM signal on pin being output in milliseconds. The minimum valid value is 1ms.
Grove Sensors		
Import from microbit module	**from mb_grove import *** **from mb_pins import ***	**TI-84 Plus CE Python Only** - Enables Grove sensor and actuator methods. Both mb_grove and mb_pins are required.

var(t),var(h) =.read_dht20()	**t,h = grove.read_dht20()**	The Grove DHT20 temperature and humidity sensor is an I2C device and must be plugged into an I2C port on any expansion card. The read_dht20() returns both temperature and humidity concurrently.
var = .read_temperature(pin)	**t = grove.read_temperature(pin0)**	The Grove Temperature sensor V1.2 is an analog device that may be used with any analog compatible pin. The .read_temperature() method returns the temperature in ◦C and requires a pin number argument.
var = .read_lightlevel(pin)	**b = grove.read_lightlevel(pin1)**	The Grove Light Sensor V1.2 is an analog device that may be used with any analog compatible pin. The .read_lightlevel() method returns the incident ambient light intensity as a percent of the sensor's maximum sensitivity (0-100%) and requires a pin number argument.
var = .read_moisture(pin)	**m = grove.read_moisture(pin2)**	The Grove Moisture Sensor V1.4 is an analog device that may be used with any analog compatible pin. The .read_moisture() method returns the contact moisture as a percent of the sensor's maximum sensitivity (0-100%) and requires a pin number argument.
var = .read_pressure(pin)	**p = grove.read_pressure(pin0)**	The Grove Integrated Pressure Sensor is an analog device that may be used with any analog compatible pin. The .read_pressure() method returns the tube connector pressure in kPa and requires a pin number argument.
.calibrate_pressure(m,b)	**grove.calibrate_pressure(.15,35)**	This method calibrates the linear output of the Grove Integrated Pressure Sensor with two calibration coefficients. These coefficients are derived from the slope and Y-intercept of a linear model relating ADC output and known tube connector pressure in kPa.
var = .read_ranger_time(pin):	**t = grove.read_ranger_time(pin0):**	The Grove Ultrasoncie Ranger V2.0 is a digital device that may be used with any digital compatible pin. The .read_ranger_time() method returns the time of flight of the ultrasonic ranger in seconds and requires a pin number argument.

var = .read_ranger_cm(pin)	**d = grove.read_ranger_cm(pin0)**	The read_ranger_cm() method works with the Grove Ultrasonic Ranger, returns the distance to an object in cm, and requires a pin number argument.
var,var,var = .read_bme280()	**t,p,h = grove.read_bme280()**	The Grove BME 280 is barometric pressure, temperature, and humidity sensor. Calculate altitude from pressure and temperature using the hypersonic formula.
var,var = .read_sgp30()	**co2,tvoc=grove.read_sgp30()**	The Grove SGP30 is an air quality sensor that measures VOC and eCO2 gasses.
.set_power(pin,pwr)	**grove.set_power(pin0,65)**	The Grove MOSFET V1.0 module is a PWM device that supplies power from external power. The .set_power() method requires two arguments, the analog output pin number and the power setting from 0% (off) to 100% (full source voltage).
.set_relay(pin,state)	**grove.set_relay(pin1,1)**	The Grove Relay V1.2 module is a digital device that switches the power to an external circuit on and off. The .set_relay() method requires two arguments, the digital output pin number and the relay state OFF(0) and ON(1).
.set_servo(pin,deg,min,max)	**grove.set_servo(pin0,45)**	The Grove Servo sweep motor is a digital device that sets the angular position of the servo. The .set_servo() method requires two arguments, the digital output pin and the angular position of the motor. Two additional optional calibration arguments set the minimum and maximum pulse train time in mSec.
Pins	**Pin0**	List of valid I/O pins. 0,1,2,8,13,14,15,16, and speaker
Data Log (TI-84 Plus CE Python Only)		
Import from microbit module	**from mb_log import ***	**TI-84 Plus CE Python Only** - Enables real-time data logging methods.
.set_duration(seconds)	**data_log.set_duration(10)**	Sets the duration of the data logging session in seconds. Each session logs 100 samples regardless of duration.
.set_sensor(sensor name)	**data_log.set_sensor('accelerometer.get_x')**	Sets the data logging sensor name. Valid sensors are built-in micro:bit sensors and any Grove sensor.
.set_range(Y-min, Y-max)	**data_log.set_range(Y-min, Y-max)**	Sets the anticipated range of the selected sensor.

.start()	**data_log.start()**	Begins collecting and graphing sensor data in real-time view. When logging is finished, the data is automatically stored in the calculator's list "L1" (time) and "L2" (sensor). The two lists may be used in the calculator's statistic application for detailed analysis and graphing.
NeoPixel		
Import from microbit module	**from mb_neopx import ***	**TI-84 Plus CE Python Only** - Enables NeoPixel methods.
color.rgb(r,g,b)	**color.rgb(255,128,0)**	A quick-access method for setting the four NeoPixel LEDs mounted on the Bit Maker expansion board. The method requires a 0-255 value for each red, green, and blue channel.
var = Color(pin)	**RGB_LED = Color(pin2)**	A constructor for use with a Grove RGB LED NeoPixel device. The method requires an object name and pin assignment.
.rgb(r,g,b)	**RGB_LED.rgb(0,255,255)**	A method of the Color class to set the red, green, and blue channels from 0->255.
np = NeoPixel(pin, pixels)	**np = NeoPixel(pin1, 20)**	A constructor for use with any NeoPixel device, such as the Grove stick or ring. Theconstructor requires a pin assignment and the number of pixels on the device. Note if the default np object name is changed, the other methods pasted from the menu must be updated to reflect the new name.
np[index] = (red,green,blue)	**np[2] = (0,255,0)**	The np object appears as a Python list. Each list element corresponds to a device pixel and must be set to a valid r,g,b value from 0->255.
.show()	**np.show()**	The np.show() illuminates the NeoPixel device according to the np[] list values.
.clear()	**np.clear()**	The np.clear() method turns off all pixels on the device.
Pins	**pin0**	A menu of valid I/O pins for the NeoPixle class.