CSPy Ulysses Documentation For Teacher And Student Usage

Paul Magnus ’18, Ines Ayara ’20, and Matthew R. Jenkins ’20, advised by Alistair Campbell

June 30, 2017

Contents

1 Introduction

This is the documentation for CSPy, a strongly-typed Python dialect for use in learning environments. This is the Ulysses version, which is the third version of the dialect. It succeeds the Jabberwocky version made by Lyndsay LaBarge '17 and Maya Montgomery '18, which is itself preceded by a foundation written by Alex Dennis '18 and Eric Collins '17. We thank those responsible for earlier versions for making our lives easier and providing a suitable framework for our improvements.

1.1 What is CSPy?

CSPy is a strongly-typed dialect of Python. Python by itself is not strongly typed and because of this, new programmers that use Python can use it in bad ways, like creating variables that end up not being used, or by setting a variable of a specific type to another type. With CSPy, we are more up front about these problems. Any things have to be named at the start of each function definition or class definition or program, and CSPy makes sure each thing is used and is not changed to a different thing by the writer. These things make it easy to teach future programmers about programming languages, and helps to make learning other object oriented programming languages easier in the future.

1.2 How To View And Use This Document

This document should give a decent primer for how to teach an introductory course in Computer Science. It discusses the grammar rules of the CSPy dialect, how to initalize variables, how to create classes, what counts as a function/procedure, what counts as an expression, what counts as a statement, and contains appendices of all built in functions and types. It is mostly meant as a reference, not as a teaching aide.

The authors advise against printing out this document, as it is lengthy and generally not designed for print form. If you have to print this document, print double sided. This document is rendered using $\LaTeX$.

1.3 Contact Information

Please send CSPy and graphical environment issues to pmagnus@hamilton.edu, any issues related to the CSPy editor to iayara@hamilton.edu, and any documentation and graphical environment issues to mjenkins@hamilton.edu. We will be happy to fix anything you find. If any of us are unable to be reached, please email acampbel@hamilton.edu and he'll forward any issues to us.

2 CSPy Grammar Rules

2.1 The Basics

Each variable declared in CSPy has to be declared in the following format, at the beginning of each function or class definition:

```
::NAME:TYPE::
```

`NAME` can be any name so long as it starts with a to z/A to Z (meaning you can't start it with a number or underscore, but you can put numbers or underscores anywhere else in the name - just like Python), but `TYPE` has to be named after an object type:

- `TYPE:int` - any integer (eg. `7`)
- `TYPE:float` - any decimal representation of a number (eg. `5.6`)
- `TYPE:bool` - `True` or `False`
- `TYPE:string` - `"Hello, world!", 'Hello, world!'`
  (NOTE: These CANNOT include new lines.)
- `TYPE:string` - `"""Hello, world!""", '''Hello, world!'''`
  (NOTE: These CAN include new lines.)
- `TYPE:list of TYPE` - `[1, 2, 3, 4]`
- `TYPE:[TYPE]` - `[1, 2, 3, 4]`
  (NOTE: The form `[TYPE]` is an alternate form, and can be useful when making a list of lists.)
- `TYPE:tuple of (TYPE * TYPE * ...)` - `("A", 1)`
- `TYPE:(TYPE * TYPE * ...)` - `("A", 1)`
  (NOTE: The form `(TYPE * TYPE * ...)` is an alternate form, and can be useful when making a tuple of tuples or a nested list of tuples.)
- `NAME:dict of [KEY | VALUE]` - `{1:"a", 2:"b"}`
- `TYPE:set of TYPE` - `{5, 6, 7, 8}`
  (NOTE: `{}` is an empty dictionary. To make an empty set, use `makeset()`.)
- `TYPE:frozenset of TYPE` - `{5, 6, 7, 8}`
  (NOTE: `{}` is an empty dictionary. To make an empty frozenset, use `frzset()`.)
- `TYPE:generator of TYPE`
  (NOTE: Generators can only be created from functions in CSPy that contain a yield statement. See 2.3.1 on page 5 for more information.)
- `TYPE:file` - Type returned by the built in function `open()`.
- `TYPE:fn (p1, p2, ...) -> r` - Function whose first parameter has type `p1`, second parameter has type `p2`, etc. and whose return type is `r`.
- `TYPE:proc (p1, p2, ...)` - Process whose first parameter has type `p1`, second parameter has type `p2`, etc. and does not return anything.

More details on types can be found in Appendix 1.

2.2 Classes

If you make a class, then the type has to be named after that class. For example, if there is a `class Circle`, an object has to be named like so:

- `NAME:Circle`

Variables of a Class type can equal an object made from that class or they can equal `None`, which means it equals no object. Only classes can equal `None`. If any other type is set to `None`, a TypeError will be thrown.

A sample declaration of `class Circle` is as follows:

```
class Circle:
    :: center:tuple of (int * int), radius:int ::
    def Circle(x:int, y:int, r:int):
        center = (x, y)
        radius = r

    def area(c:Circle) -> float:
        return (3.14 * (c.radius ** 2))
```

The variables in between the constructor and the class definition are the attributes of the class. Anything put there is only used in the scope of the class, but can be accessed through dot syntax, such as `c.center` or `c.area(d)` (assuming `c` and `d` are both of type `Circle`).

Each constructor has to be named after the class name. The constructor is only executed once (at the creation of each object), and does not return anything. In this case, `Circle(int, int, int)` is the constructor for the class `Circle`.

The method `area(c:Circle)` takes an object of type `Circle` as a parameter. Classes are allowed to have local variables in their methods whose types correspond to the class being defined.

Classes can have super classes:

```
    class Circle extends Shape:
```

In this case, `Shape` is the super class of `Circle`. All of `Shape`'s methods can be used within `Circle`.

Classes can have both constructor overloading and method overloading, so long as each formal parameter list is distinct from others in the class. For example, our `Circle` class can have two or more constructors:

```
class Circle:
    :: center:tuple of (int * int), radius:int ::
    def Circle(x:int, y:int, r:int):
        center = (x, y)
        radius = r

    def Circle(c:tuple of (int * int), r:int):
        center = c
        radius = r
    ......
```

Methods can be overloaded as well:

```
class Circle:
    :: center:tuple of (int * int), radius:int ::
    ......
    def area(c:Circle) -> float:
        return (3.14 * (c.radius ** 2))

    def area() -> float:
        return (3.14 * (radius ** 2))
```

2.3 Functions and Procedures

CSPy differentiates between Functions and Procedures. Functions must return something of the type specified, but procedures do not return anything.

The types of each variable in a formal parameter list must be declared, and if an object is returning anything it has to be indicated through `->` syntax. Anything returned must match the type after the `->`:

```
def add(x:int, y:int) -> int:
    return x + y

def add(x:float, y:float) -> float:
    return x + y
```

Function overloading is supported in CSPy, but only if each formal parameter list is distinct from others. Changing the return type does not make the function different:

```
def add(x:int, y:int) -> float:
    return tofloat(x + y)
```

If this was in the same `.cspy` file as the above two functions, the program would not run because it would detect a duplicate class and throw an exception.

Procedures can be overloaded as well, provided that each procedure's parameter list is different:

```
def output(x:int):
    print x

def output(x:float):
    print x
```

2.3.1 Generators and Yield

CSPy generators function similarly to Python generators. The type `generator` in CSPy is an iterator of arbitrary length. Within CSPy generators can only be created from a function using the `yield` statement.

```
def short_generator() -> generator of int:
    ''' This function creates a generator of a
        short list of integers '''
    for x in [4, 3, 7, 0, 3]:
        yield x
    raise StopIteration

def int_generator() -> generator of int:
    ''' This function creates a generator of all
        positive integers in sequence '''
    :: x : int = 0 ::
    while True:
        x = x + 1
        yield x
    raise StopIteration
```

All CSPy generators must end with `raise StopIteration` even if they are infinite generators as shown in the second example. As with any 'infinite' object in a computer, caution should be taken in order to avoid an unending program.

2.4 Loops and Conditionals

For loops retain the same syntax as Python. The only thing that can't be done is iteration over a tuple, because they can have values of different types within them. Being able to do this clashes with CSPy's strong typed nature, so it is better if CSPy forbids the user from doing this.

You can have a variable block within a for loop, while loop, and in conditional statements:

```
# for loop
for i in [1, 2, 3, 4]:
    :: x : int = 0 ::

# while loop
while (i in [1, 2, 3, 4]):
    :: x : int = 0 ::

# if block
if i in [1, 2, 3, 4]:
    :: x : int = 0 ::
    ......
elif i in [5, 6, 7, 8]:
    :: x : int = 1 ::
    ......
else:
    :: x : int = 2 ::
    ......
```

There are two ways to iterate in a for loop. One way is the `range()` method (see Appendix 2 for more details). CSPy introduces another form:

```
for i in 1..4:
```

The above is equivalent to the for/while loops mentioned above or to `for i in range(1, 5)`. This new form includes its last value, unlike the `range()` command. It is recommended to use the new CSPy format for iterating in a for loop instead of the range() function for this reason.

CSPy does not have the same truth testing as Python. In Python, any object can be tested for truth value (such as an empty list/dictionary/tuple/string returning False in a boolean expression). CSPy only accepts boolean values or operators/functions which return a boolean value as conditions for while loops. This is the same for conditionals (if, elif, and the ternary operator).

2.5 Statements, Expressions, and Exceptions

Statements are any of the following:

- **Assignments:**

  Variable assignment: `x = 4`

  Indexing assignment: `myList[0] = 7`

  Slicing assignment: `myList[1:] = [6, 7, 8]`

  (NOTE: Tuples can only be subject to indexing/slicing if integer literals are indices, because it is impossible to tell what type an indexing/slicing operator would return if a index is a variable. `x = myTuple[0]` is acceptable, but `x = myTuple[y]` is not.)

  Augmented assignment: `x += 10`

- **Procedure Calls:**
  `print("Hello, world!")`
- **Return Statements:**
  `return x == y` or `return`
- **Assert Statements:**
  `assert Expression` or `assert Expression, "Message"`
  (NOTE: Expression must be a boolean value or an operator/function which returns a boolean value.)
- **Raise Exceptions:**
  `raise Identifier`
  `raise Identifier("Message")`
  (NOTE: Identifier must be a built-in exception or a subclass of a built-in exception.)

  More information about built-in exceptions and extending exceptions can be found in section 2.6 on page 8.
- **Delete Statements:**
  `del Expression`
- **Special Statements:**
  `break`
  `pass`
  `continue`

Expressions are any of the following:

- **Calculations:**
  `6 + 7`
  (NOTE: Built in calculations for each type can be found at Appendix 1.)
- **Ternary Operators:**
  `(x == 1) ? 1 : 0`
  (NOTE: Python's if-else ternary operator is not supported, so CSPy uses C++ syntax. The conditional must return a boolean value or expression or a function which returns a boolean value, and each value in the then-else must be of the same type.)
- **Function Calls:**
  `add(1, 3)`
- **Indexing:**
  `x = myList[4]`, where `x` is an int and `myList` is a list of ints.
- **Slicing:**
  `s = myString[1:]`, where `s` and `myString` are strings.
- **Membership Testing:**
  `3 in [1, 2, 3]`, `3 not in [1, 2, 3]`
  (NOTE: An error will occur if the element type does not match the sequence type in the container (e.g. `4.0 in [1, 2, 3]`). This will always return False in Python and clashes with CSPy's strong typed nature, so it is better if CSPy forbids the user from doing this.)
- **Identity Testing:**
  `x is y`, `x is not y`
  (NOTE: An error will occur if you test for identity with objects with two different types. This will always return False in Python and clashes with CSPy's strong typed nature, so it is better if CSPy forbids the user from doing this.)

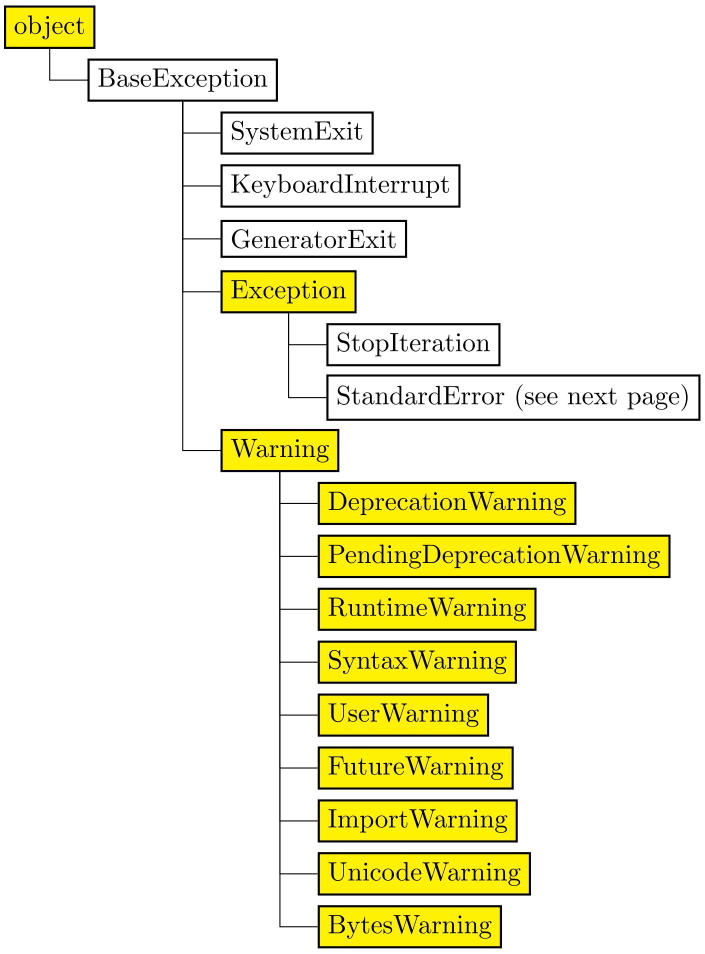
2.6 Exceptions

Exception handling uses the same syntax as Python. CSPy supports try-except, try-except-else, try-except-finally, and try-except-else-finally. Each block in a try-except clause can contain its own variable declaration block:

```
try:
    :: userInput:string, convertedUserInput:int ::
    userInput = input("Please enter a number: ")
    convertedUserInput = toint(userInput)
except TypeError:
    print("That isn’t a number.")
except ValueError as v:
    print(v)
else:
    :: userInput:string, convertedUserInput:int ::
    userInput = input("Please enter a number: ")
    convertedUserInput = toint(userInput)
finally:
    print("Hello, world!")
```

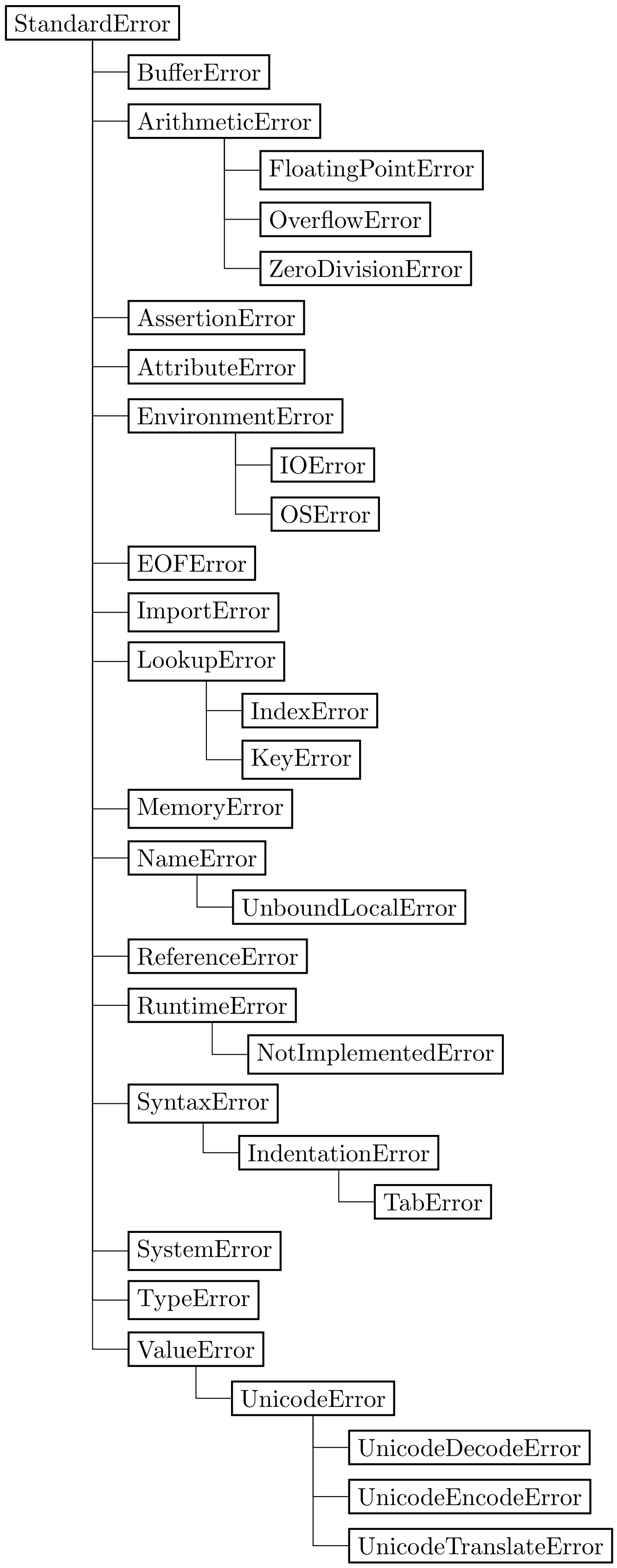
For `except` statements, if `except` is followed by an identifier, it must be a declared `Exception` type. Aliases for the caught error use the modern Python syntax `except Exception as e` when acessing the specifics of the exception are required. Calling `raise` with no arguments inside an except block will reraise the same exception that was caught by the except block.

The built-in class hierarchy for exceptions is shown below (the yellow highlighted classes can be extended by user defined classes):



All instances of exceptions can have `tostring(e)` and `repr(e)` called on them, both of which will return the message that was passed to the exception. Users should extend the `Exception` class when creating their own exceptions as shown below:

```
class MyException extends Exception:
    def MyException(message : string):
        Exception.Exception(message)
```


StandardError
BufferError
ArithmeticError
FloatingPointError
OverflowError
ZeroDivisionError
AssertionError
AttributeError
EnvironmentError
IOError
OSError
EOFError
ImportError
LookupError
IndexError
KeyError
MemoryError
NameError
UnboundLocalError
ReferenceError
RuntimeError
NotImplementedError
SyntaxError
IndentationError
TabError
SystemError
TypeError
ValueError
UnicodeError
UnicodeDecodeError
UnicodeEncodeError
UnicodeTranslateError

3 CSPy Graphics Library

3.1 The Basics

To import the library, the line "`from cs110graphics pyimport *`" needs to be the first line of your program. To put objects into the Graphics System, it requires a function which takes an object of type Window as a parameter.

```
def function(win:Window):
```

There are seven types of objects you can add to a window. Text, Image, Oval, Circle, Rectangle, Square, and Polygon. Each has its own method of initalization and requires specific parameters, but like the above function, each function requires a window object as the first parameter.

```
def function(win:Window):
    ::circ:Circle::
    circ = Circle(win, 40, (200, 200))
    win.add(circ)
```

To start the graphics system, instead of initalizing a function by calling it, you would wrap the function in a function called `StartGraphicsSystem`.

```
def function(win:Window):
    ::circ:Circle::
    circ = Circle(win, 40, (200, 200))
    win.add(circ)

StartGraphicsSystem(function)
```

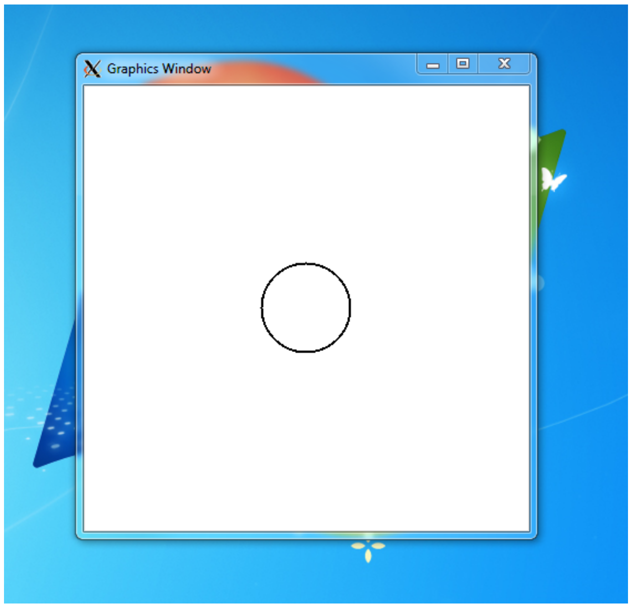


Figure 1: The above code yields a circle of radius 40 in the center of the window.

The next page contains all of the graphical objects which are included in the graphics library, as well as sample code and implementations.

- Text requires a string of text, but can optionally take a font size and a center.

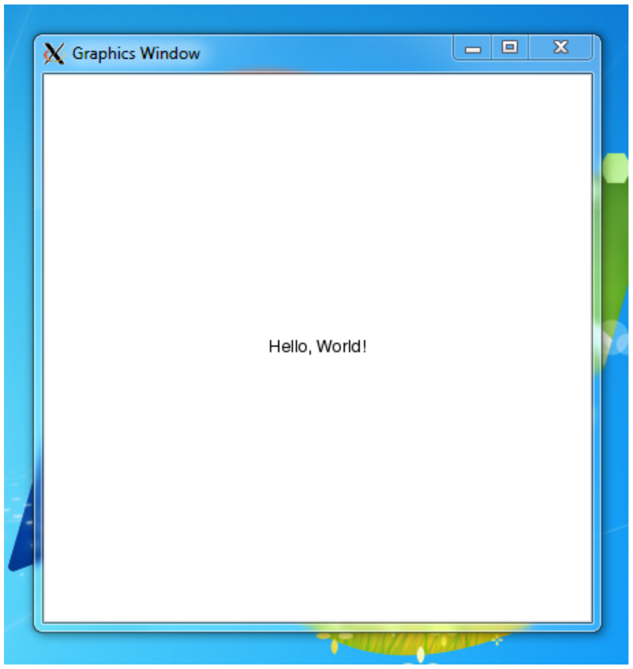


Figure 2: Text(win, "Hello, World!", 12, (200, 200))

- Image requires a name of an image, which has to be in the current working directory. It can optionally take a width, a height, and a center.

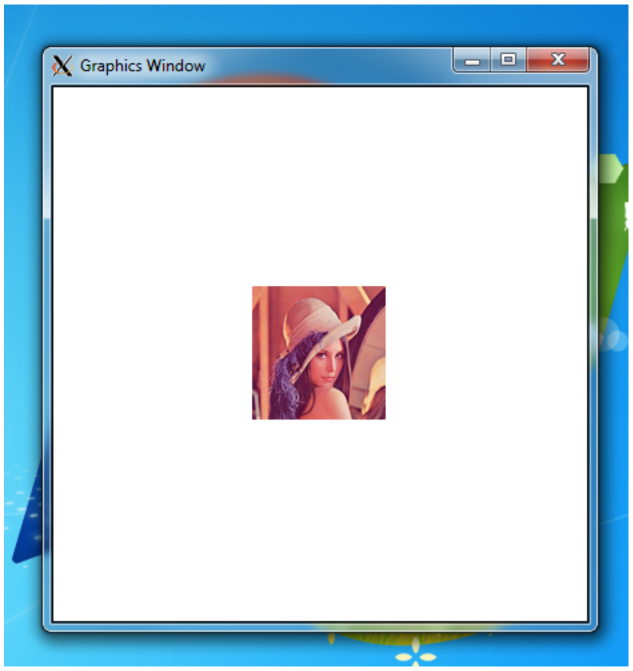


Figure 3: Image(win, "Lenna.png", 100, 100, (200, 200))

- Circles require a window, but can optionally take a radius and a center.

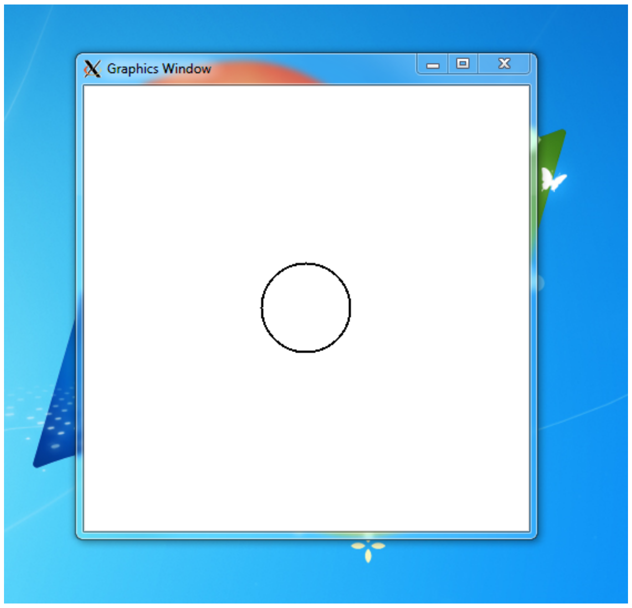


Figure 4: Circle(win, 40, (200, 200))

- Ovals require a window, but can optionally take a radiusX, a radiusY, and a center.

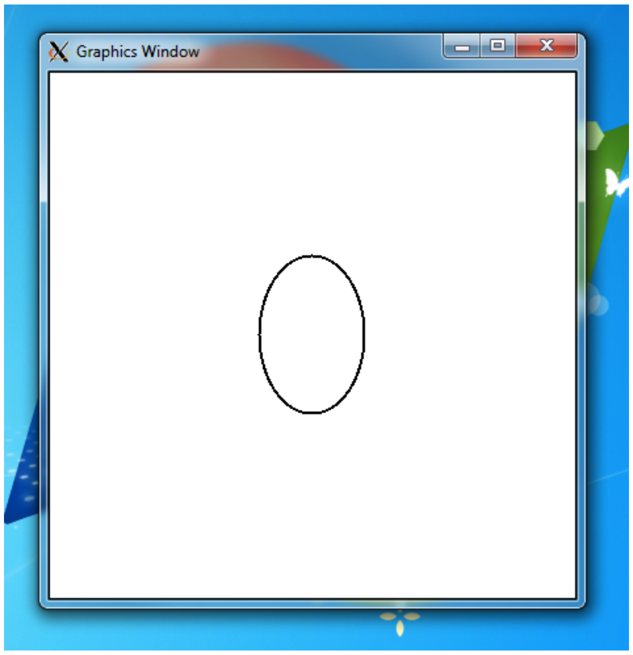


Figure 5: Oval(win, 40, 60, (200, 200))

- Squares require a window, but can optionally take a side length and a center.

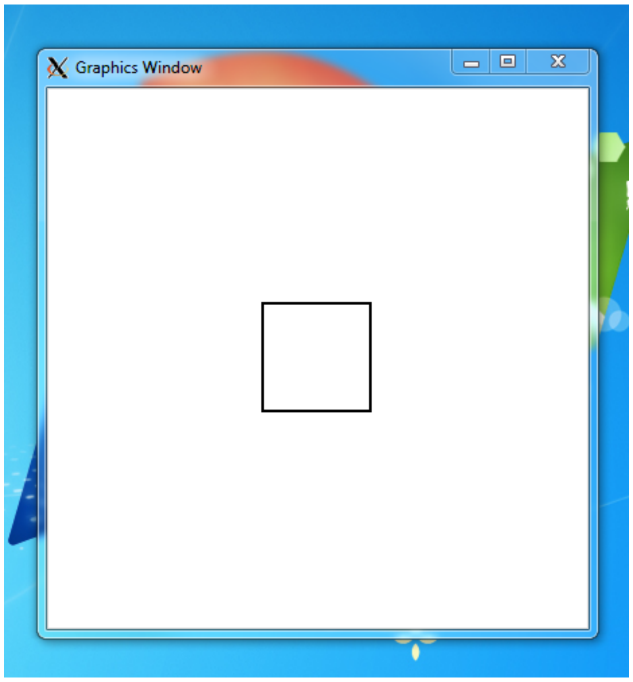


Figure 6: Square(win, 40, (200, 200))

- Rectangles require a window, but can optionally take a width, a height, and a center.

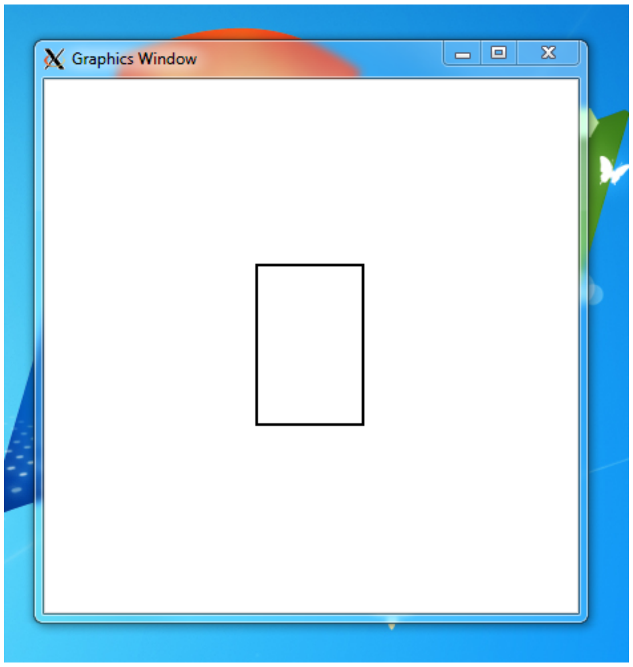


Figure 7: Rectangle(win, 40, 60, (200, 200))

- Polygons require a window and a list of points. It cannot take anything optionally.

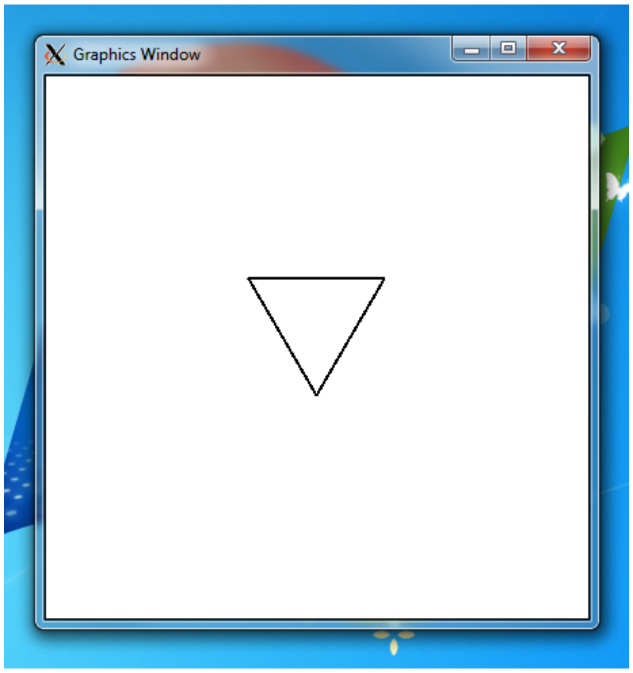


Figure 8: Polygon(win, [(150, 150), (200, 236), (250, 150)])

3.2 More Specific Methods

Objects of type `GraphicalObject` have access to `GraphicalObject` methods, and Objects of type `Fillable` have access to `Fillable` methods, as well as `GraphicalObject` methods. Some classes even have their own methods which can be accessed only by them.

`GraphicalObjects` are all seven types of object that can be put on the canvas. They have the following methods accessible:

- `add_handler(graphic:GraphicalObject)` - initalizes an EventHandler on the object and allows for overwriting of EventHandler functions by the class. (See Event Handling later in this section.)
- `get_center() -> tuple of (int * int)` - returns the center of the `GraphicalObject`.
- `get_depth() -> int` - returns the depth of the `GraphicalObject`.
- `move(dx:int, dy:int)` - Moves a `GraphicalObject` dx pixels horizontally and dy pixels vertically.
- `move_to(point:tuple of (int * int))` - moves the center of a `GraphicalObject` to the point.
- `set_depth(depth:int)` - sets the depth of the `GraphicalObject`.

`Fillables` are five of the objects that can be put on the canvas. They can have their fill colors and border colors changed, among other things. The `Circle`, `Oval`, `Rectangle`, `Square`, and `Polygon` objects are all `Fillables`.

- `get_border_color() -> string` - returns the border color of a Fillable.

- `get_border_width() -> int` - returns the border width of a Fillable.
- `get_fill_color() -> string` - returns the fill color of a Fillable.
- `get_pivot() -> tuple of (int * int)` - returns the pivot point of a Fillable.
- `rotate(degrees:int)` - rotates a Fillable by degrees.
- `scale(factor:float)` - scales a Fillable's size by the scale factor.
- `set_border_color(color:string)` - sets the border color of the Fillable.
- `set_border_width(width:int)` - sets the border width of the Fillable.
- `set_fill_color(color:string)` - sets the fill color of the Fillable.
- `set_pivot(pivot:tuple of (int * int))` - sets the pivot point of the Fillable.

You can either use names of colors like "`yellow`", or you can use hexadecimal numbers in a string like "`#FFFF00`" to set a color.

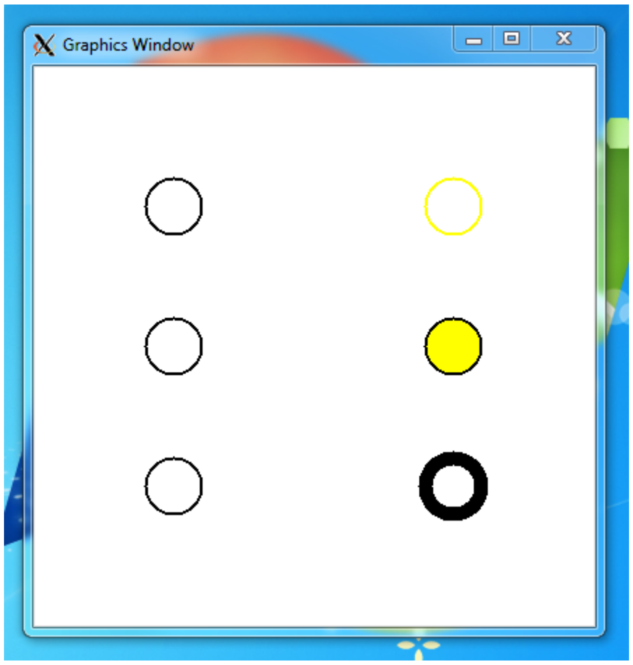


Figure 9: circ.set_border_color("yellow"), circ.set_fill_color("yellow"), circ.set_border_width(10)

Image methods (not including any inherited methods from GraphicalObject):

- `resize(width:int, height:int)` - resizes an Image by width and height.
- `rotate(degrees:int)` - rotates an Image by degrees.
- `scale(factor:float)` - scales an Image's size by scale factor
- `size() -> tuple of (int * int)` - returns a tuple of the width and height of an Image.

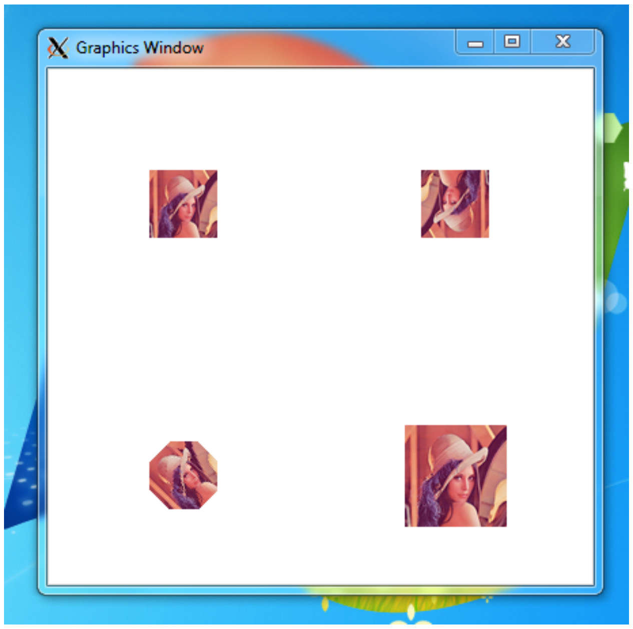


Figure 10: img.rotate(180), img.rotate(45), img.scale(1.5)

Text methods (not including any inherited methods from GraphicalObject):

- `set_text(text:string)` - Sets the text of the Text object.
- `set_size(size:int)` - sets the point size of the Text object.

Circle methods (not including any inherited methods from GraphicalObject or Fillable):

- `set_radius(radius:int)` - sets the radius of the Circle.

Oval methods (not including any inherited methods from GraphicalObject or Fillable):

- `set_radii(radiusX:int, radiusY:int)` - sets the radii of the Oval.

Square methods (not including any inherited methods from GraphicalObject or Fillable):

- `set_side_length(sideLength:int)` - sets the side length of the Square.

Rectangle methods (not including any inherited methods from GraphicalObject or Fillable):

- `set_side_lengths(width:int, height:int)` - sets the width and height of the Rectangle.

3.3 Event Handling

Event handling is the computer science term for sending keyboard and mouse commands to a graphical interface. The graphics library supports a rudimentary version of event handling.

There are two classes which do the work: `Event`, and `EventHandler`. `Event` takes an keyboard or mouse event and converts it to a format which can be read by `EventHandler`. In `Event`, you can get the location where the event occurred, the mouse button that did it, the keyboard button that did it, or a description of the event. This is useful for handling different kinds of button input.

Below is a list of Event's methods:

- `get_button() -> string` - returns the mouse button that generated the event. It will be one of the following:
  - Left Mouse Button
  - Middle Mouse Button
  - Right Mouse Button
- `get_description() -> string` - returns the description of the event. It will be one of the following:
  - Key Press
  - Key Release
  - Mouse Press
  - Mouse Release
  - Mouse Move
  - Mouse Enter
  - Mouse Leave
- `get_key() -> string` - returns the key that was pressed or released.
- `get_location() -> tuple of (int * int)` - returns the location of the mouse on the canvas.
- `get_root_location() -> tuple of (int * int)` - returns the location of the mouse on the monitor.

`EventHandler` is a class which contains what is executed when an event is sent to specific objects. It is designed so that the user can overwrite the functions in the class and replace it with their own functions.

The functions that can be overwritten are as follows:

- `handle_key_press(event:Event)` - handles a key press.
- `handle_key_release(event:Event)` - handles a key release.
- `handle_mouse_enter(event:Event)` - handles when the mouse enters an object.
- `handle_mouse_leave(event:Event)` - handles when the mouse leaves an object.
- `handle_mouse_move(event:Event)` - handles mouse movement.
- `handle_mouse_press(event:Event)` - handles a mouse press.
- `handle_mouse_release(event:Event)` - handles a mouse release.

To overwrite, the user has to extend EventHandler and initalize it in their custom class:

```
class Button extends EventHandler:
    :: circ:Circle::
    def Button(win:Window):
        EventHandler.EventHandler()
        circ = Circle(win)
        win.add(circ)
        circ.add_handler(self)
```

The user has defined a class called Button, which takes no parameters. The EventHandler is initalized, the Button's representation is made, and then the Button gets the ability to overload functions in EventHandler using the `add_handler()` function.

To overwrite a function, all you have to do is define a function that matches the names of the functions you want to overwrite. If for example, the user wants the circle's color to change when they click on it, they would do this:

```
def handleMouseRelease(event:Event):
    circ.set_fill_color("yellow")
```

After that is written by the user and run, the object's fill color will turn yellow when it is clicked.

Each function requires an object of type Event to be attached so that if the user wants to know more details about the EventHandler, they can access the Event methods discussed above. For example, what if the user wants to know where the mouse was clicked within an object? The user would then use the Event class and call getMouseLocation() to find out:

```
def handleMouseRelease(event:Event):
    print(event.getMouseLocation())
```

3.4 Animations

There are two ways to do animations. One is to use the Timer class. When initalized, a Timer takes a window, a delay (in milliseconds), and a function. The timer will re-run the function after each delay of time. To start the timer, call the `start()` function. To stop the timer, call the `stop()` function.

```
class Button:
    :: circ:Circle, timer:Timer, pressed:bool ::
    def Button(win:Window):
        circ = Circle(win)
        win.add(circ)
        timer = Timer(win, 200, flash)
        pressed = False
        timer.start()

    def flash():
        if pressed:
            circ.set_fill_color("")
            pressed = False
        else:
            circ.set_fill_color("yellow")
            pressed = True
```

This code example has several parts to it. It first initalizes a Circle, a Timer and a Boolean. It creates the circle, adds it, creates the timer, sets the boolean to False, then starts the timer. The timer then runs the function `flash`, which sets the fill color to yellow, then after 200 seconds sets it to be transparent.

Another way to do animations is to run an instance of the `RunWithYieldDelay` class. This class takes a function which has the CSPy keyword `yield` and then allows that function to run with a delay.

```
def main(win:Window):
    :: circ:Circle ::
    circ = Circle(win)
    win.add(circ)
    RunWithYieldDelay(win, move_circle(circ))
```

```
def move_circle(circ:Circle) -> generator of int:
    for i in range(10):
        circ.move(10, 0)
        yield 200
    raise StopIteration
```

This function will keep running until the for loop stops.

4 Appendix 1: Types

Int: A numeric type.
Binary Operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Floor Division	//
Exponentiation	**
Modulus	%
Bitwise And	&
Bitwise Or	|
Bitwise Xor	^
Bitwise Invert	~
Equals	==
Not Equals	!=
Greater Than	>
Less Than	<
Greater Or Equal To	>=
Less Or Equal To	<=
Bitwise Left Shift	<<
Bitwise Right Shift	>>

Unary Operators:

Positive	+
Negative	-

Type Conversion:

- Float via the `tofloat` built in function
- String via the `tostring` or `repr` built in function

Additional Operations:

- Supports augmented assignment.

Float: A numeric type.

Binary Operators:

Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Floor Division	`//`
Exponentiation	`**`
Modulus	`%`
Equals	`==`
Not Equals	`!=`
Greater Than	`>`
Less Than	`<`
Greater Or Equal To	`>=`
Less Or Equal To	`<=`

Unary Operators:

Positive	+
Negative	`-`

Type Conversion:

- Int via the `toint` or `round` built in function
- String via the `tostring` or `repr` built in function

Additional Operations:

- Supports augmented assignment.

Bool:
Binary Operators:

Boolean And	`and`
Boolean And	`&&`
Boolean Or	`or`
Boolean Or	`||`

Unary Operators:

Boolean Not	`not`

Type Conversion:

- String via the `tostring` or `repr` built in function

(NOTE: In Python, bool is a subclass of Integer, therefore Integer binary operators such as '+', '-', etc. would be applicable to Boolean values as well. This is not the case in CSPy, as there is no benefit for a beginner programmer to use any of these operators on boolean.)

String: A sequence type.

Binary Operators:

Concatenate	`+`
Repetition	`*`
Greater Than	`>`
Less Than	`<`
Greater Or Equal To	`>=`
Less Or Equal To	`<=`
Equals	`==`
Not Equals	`!=`

Type Conversion:

- Integer via `toint` built-in function
- Float via `tofloat` built-in function
- List, Set, or Frozenset via `tolist` or `makeset` or `frzset` built-in functions

Additional Operations:

- String supports indexing, slicing, membership, iterations, and use of the `len()` function. Also supports augmented assignment.

List: A sequence type.

Binary Operators:

Concatenate	`+`
Repetition	`*`
Equals	`==`
Not Equals	`!=`

Type Conversion:

- String via `tostring` or `repr` built-in function
- Set or Frozenset via `makeset` or `frzset` built-in functions
- Bool via `tobool` built-in function

Additional Operations:

- List supports indexing, slicing, membership, iterations, and use of the `len()` function. Also supports augmented assignment.

Tuples: A sequence type. Tuples are heterogeneous, and can contain elements of multiple types. Tuples are also immutable and can't be changed once created.

Binary Operators:

Concatenate	`+`
Repetition	`*`
Equals	`==`
Not Equals	`!=`

Type Conversion:

- String via `tostring` or `repr` built-in function
- List, Set or Frozenset via `tolist`, `makeset` or `frzset` built-in functions (only applicable for homogeneous tuples)

Additional Operations:

- Tuples support indexing, slicing, membership testing, iteration, and use of the `len()` function. Tuples do not support augmented assignment (they are immutable).

(NOTE: The methods for tuples have generic object return types because the return type of these operators depends on the type of the tuple. Since tuples can contain multiple types unlike lists, the return type of a slicing or indexing operation is determined in the type checker by analyzing the elem_type list of the tuple. Because it is impossible to determine the return type of an indexing or slicing operation of a multi-typed tuple using a variable as an index, indexing tuples with variables is NOT allowed.

`mytuple[2] ->` acceptable
`mytuple[p] ->` unacceptable

Similarly, because it is impossible to determine the type of an iterative variable in a for loop iterating over a multi-type tuple, tuples DO NOT support iterating.)

Dictionary: An associative array.

Binary Operators:

Equals	`==`
Not Equals	`!=`

Type Conversion:

- String via `tostring` or `repr` built-in function
- List, Set or Frozenset via `tolist`, `makeset` or `frzset` built-in functions
- Bool via `tobool` built-in function

Additional Operations:

- Dictionaries support indexing, slicing, membership testing, iteration, and use of the len() function.

Sets and Frozensets:
An unordered collection of unique elements. Sets and frozensets are homogeneous, i.e. they can only contain one element type. Sets are mutable but frozensets are immutable.

Binary Operators:

Proper Subset	`<`
Proper Superset	`>`
Superset	`<=`
Superset	`>=`
Union	`|`
Intersection	`&`
Difference	`-`
Symmetric Difference	`^`
Not Equal	`!=`
Equals	`==`

Type Conversion:

- String via `tostring` or `repr` built-in function
- List, Set or Frozenset via `tolist`, `makeset` or `frzset` built-in functions
- Bool via `tobool` built-in function

Additional Operations:

- Sets and frozensets support membership testing and iteration. Sets support augmented assignment. Frozensets do not (they are immutable).

Functions:
A function is a procedure that returns a value. Functions have a return type. All of the return statements in a function must be nonempty and their return value must have the same type as the functions return type.

CSPy supports function overloading, provided that each function or procedure has a distinct parameter list. (NOTE: See CSPy Grammar Rules - Functions and Procedures for more details.)

A declared variable may be bound to a function:

```
f:fn (int, int) -> int = add
```

The above will assign the overloaded add function corresponding to the given type signature to f. Similarly, the below example is also valid because the type of the anonymous function matches the type of f:

```
f:fn (int, int) -> int = lambda (x:int, y:int) -> int : (x + y)
```

(NOTE: `fn (int, ?int) -> int` is not equivalent to `fn (int, int) -> int`. The `?` symbol in the first function type indicates the second integer is an optional parameter. This is the same for procedures.)

Procedures:
A procedure does not return a value and hence has no return type. Procedures support function overloading as well. (NOTE: See CSPy Grammar Rules - Functions and Procedures for more details.)

Declared variables may be bound to procedures using the following syntax:

```
p:proc (int) = output
```

Files:
Files are the type for a Python file object. A file is created by using the built-in open() function. Attributes of the file type are as follows:

- `closed:bool` - True if the file is closed, False otherwise.
- `name:string` - The name of the file.
- `mode:string` - The mode which the file was opened with.

5 Appendix 2: Built In Functions

Python Built Ins: (NOTE: Not all of the built-in Python functions are currently implemented in CSPy. The below functions have been implemented.)

- `abs(x:int) -> int`
  `abs(x:float) -> float`
  Returns the absolute value of x.
- `all(l:list of ?) -> bool`
  Returns true if all the elements in l are true.
- `any(l:list of ?) -> bool`
  Returns true if any of the elements in l are true.
- `bin(x:int) -> string`
  Converts x into a binary string.
- `chr(i:int) -> string`
  Returns a string representing a character whose Unicode point is i.
- `cmp(a:int, b:int) -> int`
  `cmp(a:float, b:float) -> int`
  `cmp(a:string, b:string) -> int`
  `cmp(a:bool, b:bool) -> int`
  `cmp(a:list, b:list) -> int`
  `cmp(a:tuple, b:tuple) -> int`
  `cmp(a:dict, b:dict) -> int`
  `cmp(a:set, b:set) -> int`
  `cmp(a:frozenset, b:frozenset) -> int`
  Returns 1 if a > b, -1 if a < b, and 0 if a == b.
- `divmod(a:int, b:int) -> tuple of (int * int)`
  `divmod(a:float, b:float) -> tuple of (float * float)`
  Returns a pair of numbers consisting of the quotient of a and b and their remainder when using integer division. For integers, this is equivalent to `(a // b, a % b)`. For floats, this is equivalent to `(math.floor(a / b), a % b)`.
- `exit(?code:int)`
  Exits from the current program.
- `hex(x:int) -> string`
  Converts x to a lowercase hexadecimal string prefixed with '0x'.
- `len(s:string) -> int` `len(l:list) -> int`
  `len(t:tuple) -> int`
  `len(d:dict) -> int`
  `len(mset:set) -> int`
  `len(fr:frozenset) -> int`
  Returns the number of objects in the given sequence or container.
- `max(a:int, b:int) -> int` `max(a:float, b:float) -> float`
  `max(l:list of elem_type) -> elem_type`
  For integers and floats, returns a if a > b or b if b > a. For lists, returns the item from the list with max value.

- `map(f:func, l:list) -> list` `map(f:func, d:dict) -> dict`
  `map(f:func, t:tuple) -> tuple`
  `map(f:func, mset:set) -> set`
  `map(f:func, fr:frozenset) -> frozenset`
  Applies the function f to each value within the given sequence or container and returns a new container.

- `min(a:int, b:int) -> int` `min(a:float, b:float) -> float`
  `min(l:list of elem_type) -> elem\_type`
  For integers and floats, returns a if $a < b$ or b if $b < a$. For lists, returns the item from the list with min value.

- `oct(x:int) -> string`
  Converts x to an octal string.

- `ord(s:string) -> int`
  Given a string representing a Unicode character s, returns an integer representing the Unicode point of s.

- `open(name:string, mode:string = "r") -> file`
  Open the file name in mode. If mode is not given, defaults to r (read). Returns a file object.

- `pow(x:int, y:int) -> int`
  `pow(x:int, y:int, z:int) -> int`
  Returns x to the power y. If z is present, returns x to the power y modulo z.

- `range(stop:int) -> list of int`
  `range(start:int, stop:int, step:int = 1 ) - > list of int`
  Returns a list of integers representing the range of integers from start to stop using step if given. If only stop is given, start defaults to 0.

- `round(x:float, y:int = 0) -> int`
  Returns x rounded to y digits after the decimal point. If y is omitted, returns the nearest integer to its input.

- `sum(l:list of int, start:int = 0) -> int`
  `sum(l:list of float, start:float = 0.0) -> float`
  `sum(t:tuple of int, start:int = 0) -> int`
  `sum(t:tuple of float, start:float = 0.0) -> float`
  `sum(t:set of int, start:int = 0) -> int`
  `sum(t:set of float, start:float = 0.0) -> float`
  `sum(t:frozenset of int, start:int = 0) -> int`
  `sum(t:frozenset of float, start:float = 0.0) -> float`
  Sums start and the items of the iterable from left to right and returns the total. start defaults to 0 if not given.

String Built Ins:

- `capitalize() -> string`
  Capitalizes a string.

- `center(width:int, ?fillchar:string =  ) -> string`
  Returns a centered string of length width whose padding is done using the specified fill character. The default fill character is a space.

- `count(str:string, ?beg:int = 0, ?end: int = len(string)) -> int`
  Counts how many times str occurs in string or in a substring of the current string if the starting index beg and ending index end are given.

- `decode(?encoding = UTF-8, ?errors = strict) -> string`
  Decodes the current string using the codec encoding, which defaults to the default string encoding. Errors is the error handling scheme, which defaults to strict, meaning encoding errors will raise a UnicodeError.

- `encode(?encoding = UTF-8, ?errors = strict) -> string`
  Encodes the current string using the codec encoding, which defaults to the default string encoding. Errors if the error handling scheme, which defaults to strict, meaning encoding errors will raise a UnicodeError.

- `endswith(suffix:string, ?beg:int = 0, ?end:int = len(string)) -> bool`
  Determines whether or not the current string ends with suffix (or a substring of a string if starting index beg and ending index end are given).

- `expandtabs(?tabsize:int = 8) -> string`
  Expands tabs in string to multiple spaces. Defaults to 8 spaces per tab if tabsize is not provided.

- `find(str:string, ?beg:int = 0, ?end:int = len(string)) -> int`
  Determines if str occurs in the current string or in a substring if starting index beg and ending index end are given. Returns starting index of str if found, else returns -1.

- `index(str:string, ?beg:int = 0, ?end:int = len(string)) -> int`
  Same as find, but raises an exception is str is not found.

- `isalnum() -> bool`
  Returns true if string has at least 1 character and all the characters are alphanumeric.

- `isalpha() -> bool`
  Same as isalnum.

- `isdigit() -> bool`
  Returns true if the string contains only digits.

- `islower() -> bool`
  Returns true if string has at least 1 cased character and all cased characters are in lowercase.

- `isnumeric() -> bool`
  Returns true if a Unicode string contains only numeric characters.

- `isspace() -> bool`
  Returns true if the string contains only whitespace characters.

- `istitle() -> bool`
  Returns true if the string is properly titlecased.

- `isupper() -> bool`
  Returns true if string has at least one cased character and all cased characters are in uppercase.

- `join(seq:list/tuple of string) -> string`
  Concatenates the elements in the sequence into a string with the current string as a separator. The elements in the sequence must be strings.

- `ljust(width:int, ?fillchar:string =  ) -> string`
  Returns a left justified string of length width whose padding is fillchar, which defaults to a space.

- `lower() -> string`
  Converts all uppercase letters to lowercase.
- `lstrip() -> string`
  Removes all leading whitespace in the current string.
- `replace(old:string, new:string, max:int) -> string`
  Returns a copy of the string with all occurrences of the substring old replaced by new if max is not specified. If max is specified, only max occurrences will be replaced starting from the front of the string.
- `rfind(str:string, ?beg:int = 0, ?end:int = len(string)) -> int`
  Same as find(), but searches backwards in the string.
- `rindex(str:string, ?beg:int = 0, ?end:int = len(string)) -> int`
  Same as index(), but searches backwards in the string.
- `rjust(width:int, ?fillchar:string =  ) -> string`
  Returns the original string right justified to a total width of columns using fillchar, which defaults to a space.
- `rstrip() -> string`
  Removes all of the trailing whitespace on a string.
- `split(str:string =  , ?num:int = string.count(str)) -> list of string`
  Splits strings according to str (defaults to a space) and returns a list of substrings. Splits into at most num substrings if num is given.
- `splitlines(?num:int = string.count(\n)) -> list of string`
  Splits at all (or num if given) new lines and returns a list of each line with newlines removed.
- `startswith(str:string, ?beg:int = 0, ?end:int = len(string)) -> bool`
  Determines if the current string (or a substring of string if the starting index beg and ending index end are given) starts with the substring str.
- `strip() -> string`
  Performs both lstrip() and rstrip() at the same time.
- `swapcase() -> string`
  Inverts the case for all letters in a string.
- `title() -> string`
  Returns titlecased version of the current string where all words begin with uppercase letters and the rest are lowercase.
- `upper() -> string`
  Converts lowercase letters in the current string to uppercase.

List Built Ins:

- `append(obj:elem_type)`
  Appends obj to the end of the list.
- `count(obj:elem_type) -> int`
  Returns count of how many times obj occurs in list.

- `extend(seq:list of elem_type)`
  Appends the contents of seq to list.
- `index(obj:elem_type) -> index`
  Returns the first index in list where obj appears.
- `insert(index:int, obj:elem_type)`
  Inserts obj into list at offset index.
- `pop(?index:int = -1) -> elem_type`
  Removes and returns the object at index from list, or the end of the list if index was not given.
- `remove(obj:elem_type)`
  Removes obj from list.
- `reverse()`
  Reverses the order of the objects in the list.
- `sort()`
  Sorts the objects in the list.

Dictionary Built Ins:

- `clear()`
  Removes all elements of the current dictionary.
- `copy() -> dict of [key_type|value_type]`
  Returns a shallow copy of the current dictionary.
- `get(key:key_type, default:value_type) -> value_type`
  For key, returns its value, or default if key not in dictionary.

  (NOTE: In Python, default defaults to None type, but since None is for classes only, default is required.)
- `has_key(key:key_type) -> bool`
  Returns true if key is in dictionary.
- `items() -> list of tuple of (key_type * value_type)`
  Returns a list of the current dictionarys (key, value) tuple pairs.
- `keys() -> list of key_type`
  Returns list of the current dictionarys keys.
- `pop(elem:key_type, ?default:value_type) -> value_type`
  If elem is in the current dictionary, removes elem from the dictionary and returns its value. If elem is not in the current dictionary and default was not given, raises KeyError.
- `popitem() -> tuple of (key_type * value_type)`
  Removes and returns an arbitrary (key, value) pair from the current dictionary. If the dictionary is empty, calling popitem() raises a KeyError.
- `setdefault(key:key_type, v:value_type)`
  Sets dict[key] = v if key is not already in the current dictionary.

  (NOTE: In Python, v defaults to None type, but since None is for classes only, 'v' is required.)
- `update(dict2:dict of [key_type|value_type])`
  Adds dictionary dict2s key-value pairs to the current dictionary.

- `values() -> list of value_type`
  Returns a list of the values in the current dictionary.

Set/Frozenset Built Ins:

(NOTE: The function signatures of set and frozenset methods depend on the element type, like lists and dictionaries. There are init_set and init_frzset functions that generate a typed method dictionary for their respective types.)

- `isdisjoint(s:set/frzset of elem_type) -> bool` Returns true if the current set is disjoint from s (the set has no elements in common with s).
- `issubset(s:set/frzset of elem_type) -> bool` Returns true if the current set is a subset of s (every element of the current set is in s).
- `issuperset(s:set/frzset of elem_type) -> bool` Returns true if the current set is a superset of s (every element of s is in the current set).
- `union(s:set/frzset of elem_type) -> set/frzset of elem_type` Returns a new set that is the union of the current set and s (a set containing all elements from current set and s).
- `intersection(s:set/frzset of elem_type) -> set/frzset of elem_type` Returns a new set that is the intersection of the current set and s (a set with all elements that are in both the current set and s).
- `difference(s:set/frzset of elem_type) -> set/frzset of elem_type` Returns a new set with all elements in the current set that are not in s.
- `symmetric_difference(s:set/frzset of elem_type) -> set/frzset of elem_type` Returns a new set with all elements in either the current set or s but not both.
- `copy() -> set/frzset of elem_type` Returns a shallow copy of the current set.

Set ONLY Built Ins - do not apply to frozenset:

- `update(s:set)` Update the current set by adding all elements from set s.
- `intersection_update(s:set):` Update the current set by keeping only elements found in both the current set and the set s.
- `difference_update(s:set):` Update the current set by keeping only elements found in either the current set or s, but not in both.
- `add(elem:elem_type)` Add the element elem to the current set.
- `remove(elem:elem_type)` Removes element elem from the current set. Raises KeyError if elem is not in the current set.
- `discard(elem:elem_type)` Same as remove but does not raise KeyError if elem is not present in the current set.
- `pop() -> elem_type` Removes and returns an arbitrary element from the current set. Raises KeyError if the set is empty.
- `clear()` Removes all elements from the current set.

File Built Ins:

- `close()`
  Closes the file. A closed file cannot be read or written to anymore. Any operation which requires that the file be open will raise a ValueError is the file is closed. Calling close more than once is allowed.

- `flush()`
  Flushes the internal buffer.

- `fileno() -> int`
  Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.

- `next() -> string`
  Returns the next line from the file each time it is being called.

- `read(?size:int = file size) -> string`
  Read at most size bytes from the current file, less if hits EOF before reaching size bytes. If size is not given, reads the entire file.

- `readline(?size:int = file size) -> string`
  Reads one line from the file. If the size argument is present, it is a maximum byte count of the line. An empty string is returned only when EOF is encountered immediately.

- `readlines(?size:int = file size) -> list of strings`
  Reads until EOF using readline and return a list containing the lines. If size is given, instead of reading up to EOF, reads whole lines totaling approximately size bytes in size.

- `seek(offset:int, ?whence:int = 0)`
  Sets the current file position to offset. If whence is given, sets the current position to the offset from whence.

- `tell() -> int`
  Returns the files current position.

- `truncate(?size:int = ?)`
  Truncates the file size. If size is given, the file is truncated to at most that size.

- `write(str:string)`
  Writes str to the current file.

- `writelines(seq:list of string)`
  Writes a sequence of strings from a list to the current file.