Rust → Python Quick Reference

Essential Syntax Conversions

Basic Syntax

Feature	Rust	Python
Comments	// Single line /* Multi line */ /// Doc comment	# Single line """ Multi line """ # No special doc syntax

Variables	let x = 5;
	let mut y = 10; // mutable
		x = 5
		y = 10 # always mutable

Constants	const MAX: i32 = 100;	MAX = 100 # or MAX: Final[int] = 100

Print	println!("Hello {}", name);
	print!("Hello");
		print(f"Hello {name}")
		print("Hello", end="")

Macros	println!() vec![] format!()	N/A - functions instead
Statement end	Semicolon required ;	No semicolon needed
Blocks	Braces { }	Indentation (4 spaces)
Expressions	Everything is expression let x = if y { 1 } else { 2 };	Most things are expressions x = 1 if y else 2

Variables & Types

Type	Rust	Python
Integer	let x: i32 = 42; let y: u64 = 100; // unsigned	x = 42 # Python int is arbitrary precision

Float	let x: f64 = 3.14;
	let y: f32 = 2.5;
		x = 3.14
		# Python float is f64

Boolean	let flag: bool = true;
	let flag = false;
		flag = True
		flag = False

String	let s: String = String::from("text");
	let s = "text".to_string();
		s = "text"
		# Always Unicode, always heap

String slice	let s: &str = "text";
	let slice = &s[0..4];
		s = "text"
		slice = s[0:4]

Char	let c: char = 'a'; // 4 bytes Unicode	c = "a" # just a string
Array	let arr: [i32; 3] = [1, 2, 3]; // Fixed size, stack	arr = [1, 2, 3] # Dynamic list, heap

Tuple	let t: (i32, f64) = (1, 2.5);
	let (x, y) = t;
		t = (1, 2.5)
		x, y = t

Option	let x: Option<i32> = Some(5);
	let y: Option<i32> = None;
		x = 5 # or None
		from typing import Optional
		x: Optional[int] = 5

Type check	if let Some(val) = x { } matches!(x, Some(_))	isinstance(x, int) type(x) == int

Type cast	x as f64
	x.into()
		float(x)
		int(x)

Ownership & Borrowing (Rust-Specific)

Concept	Rust	Python Equivalent

Ownership	let s1 = String::from("hello");
	let s2 = s1; // s1 invalid now
	// Move semantics
		s1 = "hello"
		s2 = s1 # Both valid
		# Reference counting

Clone	let s1 = String::from("hello");
	let s2 = s1.clone(); // Deep copy
		import copy
		s2 = copy.deepcopy(s1)

Reference	fn read(s: &String) { }
	read(&my_string); // Borrow
		def read(s):
		    pass
		read(my_string) # Always ref

Mut reference	fn modify(s: &mut String) { }
	modify(&mut my_string);
		def modify(s):
		    s.append(x)
		modify(my_list) # Mutable

Lifetimes	fn longest<'a>(x: &'a str, y: &'a str) -> &'a str	def longest(x, y):
		    # No lifetime needed

Drop	impl Drop for MyType {
	    fn drop(&mut self) { }
	}
		def __del__(self):
		    # Cleanup

String Operations

Operation	Rust	Python

Create	let s = String::from("text");
	let s = "text".to_string();
		s = "text"

String slice	let s: &str = "text";
	let slice = &s[0..4];
		s = "text"
		slice = s[0:4]

Concatenation	let s = format!("{} {}", s1, s2);
	let s = s1 + &s2; // moves s1
		s = s1 + " " + s2
		s = f"{s1} {s2}"

Length	s.len() // bytes
	s.chars().count() // chars
		len(s) # characters

Is empty	s.is_empty()	not s len(s) == 0

Upper/Lower	s.to_uppercase()
	s.to_lowercase()
		s.upper()
		s.lower()

Trim	s.trim()
	s.trim_start()
	s.trim_end()
		s.strip()
		s.lstrip()
		s.rstrip()

Replace	s.replace("old", "new")	s.replace("old", "new")
Split	s.split(",").collect::<Vec<_>>()	s.split(",")
Join	vec.join(",")	",".join(list)
Contains	s.contains("text")	"text" in s
Parse	let n: i32 = s.parse().unwrap();	n = int(s)
Format	format!("Value: {}", x) format!("Hex: {:x}", x)	f"Value: {x}" f"Hex: {x:x}"

Collections (Vec, HashMap)

Operation	Rust	Python

Vector	let mut v = Vec::new();
	let v = vec![1, 2, 3];
		v = []
		v = [1, 2, 3]

HashMap	use std::collections::HashMap;
	let mut m = HashMap::new();
	m.insert("key", "val");
		m = {}
		m = {"key": "val"}
		m["key"] = "val"

Access	v[0] // panics if out of bounds v.get(0) // returns Option	v[0] # raises IndexError m["key"] # raises KeyError

Safe access	match v.get(0) {
	    Some(x) => ...,
	    None => ...,
	}
		x = v[0] if len(v) > 0 else None
		x = m.get("key")

Push	v.push(4);	v.append(4)
Pop	v.pop() // returns Option<T>	v.pop() # raises if empty

Insert	v.insert(0, val);	v.insert(0, val)
Remove	v.remove(0);	v.pop(0) del v[0]
Length	v.len()	len(v)
Is empty	v.is_empty()	not v len(v) == 0

Contains	v.contains(&val)
	m.contains_key(&key)
		val in v
		"key" in m

Iterate	for item in &v { }
	for (k, v) in &m { }
		for item in v:
		    pass
		for k, v in m.items():

Map	v.iter().map(|x| x * 2).collect()	[x * 2 for x in v] map(lambda x: x*2, v)
Filter	v.iter().filter(|x| x > 5).collect()	[x for x in v if x > 5] filter(lambda x: x>5, v)

Control Flow

Structure	Rust	Python

If/Else	if x > 0 {
	    // code
	} else if x == 0 {
	    // code
	} else {
	    // code
	}
		if x > 0:
		    # code
		elif x == 0:
		    # code
		else:
		    # code

If expression	let y = if x > 0 { 1 } else { 0 };	y = 1 if x > 0 else 0

Loop	loop {
	    if done { break; }
	}
		while True:
		    if done:
		        break

While	while x < 10 {
	    x += 1;
	}
		while x < 10:
		    x += 1

For range	for i in 0..10 {
	    // 0 to 9
	}
	for i in 0..=10 {
	    // 0 to 10
	}
		for i in range(10):
		    # 0 to 9
		for i in range(11):
		    # 0 to 10

For each	for item in &vec {
	    // immutable ref
	}
	for item in &mut vec {
	    // mutable ref
	}
		for item in list:
		    # code

Enumerate	for (i, item) in vec.iter().enumerate() {
	    // code
	}
		for i, item in enumerate(list):
		    # code

Break/Continue	break;
	continue;
		break
		continue

Loop labels	'outer: loop {
	    loop {
	        break 'outer;
	    }
	}
		# Use flag variable or
		# nested function with return

Pattern Matching

Pattern	Rust	Python

Match basic	match x {
	    1 => println!("one"),
	    2 => println!("two"),
	    _ => println!("other"),
	}
		match x: # Python 3.10+
		    case 1:
		        print("one")
		    case 2:
		        print("two")
		    case _:
		        print("other")

Match range	match x {
	    1..=5 => "low",
	    6..=10 => "high",
	    _ => "other",
	}
		match x:
		    case x if 1 <= x <= 5:
		        "low"
		    case x if 6 <= x <= 10:
		        "high"

Match Option	match opt {
	    Some(val) => val,
	    None => 0,
	}
		if opt is not None:
		    val = opt
		else:
		    val = 0
		# or: val = opt or 0

Match Result	match result {
	    Ok(val) => val,
	    Err(e) => panic!("{}", e),
	}
		try:
		    val = result()
		except Exception as e:
		    raise

If let	if let Some(x) = opt {
	    // use x
	}
		if opt is not None:
		    x = opt
		    # use x

While let	while let Some(x) = iter.next() {
	    // use x
	}
		for x in iter:
		    # use x
		# or check with walrus

Destructure	let (x, y, z) = tuple;
	match point {
	    Point { x, y } => ...,
	}
		x, y, z = tuple
		# dataclass/namedtuple:
		x, y = point.x, point.y

Functions

Concept	Rust	Python

Basic	fn add(a: i32, b: i32) -> i32 {
	    a + b // no semicolon = return
	}
		def add(a, b):
		    return a + b
		# or
		def add(a: int, b: int) -> int:

No return	fn print_it(s: &str) {
	    println!("{}", s);
	}
		def print_it(s):
		    print(s)

Multiple return	fn get_pair() -> (i32, i32) {
	    (1, 2)
	}
		def get_pair():
		    return 1, 2
		# or return (1, 2)

Default args	// Not supported directly
	// Use Option or builder pattern
		def func(a, b=10):
		    return a + b

Closure	let add = |x, y| x + y;
	let square = |x| x * x;
		add = lambda x, y: x + y
		square = lambda x: x * x

Closure capture	let x = 10;
	let add_x = |y| x + y; // borrows x
		x = 10
		add_x = lambda y: x + y

Move closure	let x = vec![1, 2, 3];
	let take_x = move || x;
		# Python closures always capture
		x = [1, 2, 3]
		take_x = lambda: x

Higher-order	fn apply<F>(f: F, x: i32) -> i32
	where F: Fn(i32) -> i32
	{
	    f(x)
	}
		def apply(f, x):
		    return f(x)

Method syntax	impl MyType {
	    fn method(&self) -> i32 {
	        self.value
	    }
	}
		class MyType:
		    def method(self):
		        return self.value

Structs, Traits & Enums

Concept	Rust	Python

Struct	struct Point {
	    x: i32,
	    y: i32,
	}
	let p = Point { x: 0, y: 0 };
		from dataclasses import dataclass
		@dataclass
		class Point:
		    x: int
		    y: int
		p = Point(0, 0)

Methods	impl Point {
	    fn distance(&self) -> f64 {
	        ((self.x.pow(2) + self.y.pow(2)) as f64).sqrt()
	    }
	}
		class Point:
		    def distance(self):
		        return (self.x**2 + self.y**2)**0.5

Trait	trait Draw {
	    fn draw(&self);
	}
	impl Draw for Circle {
	    fn draw(&self) { }
	}
		from abc import ABC, abstractmethod
		class Draw(ABC):
		    @abstractmethod
		    def draw(self):
		        pass

Derive	#[derive(Debug, Clone, PartialEq)]
	struct Point { x: i32 }
		@dataclass
		class Point:
		    x: int
		# Auto gets __repr__, __eq__

Enum	enum Color {
	    Red,
	    Green,
	    Blue,
	}
		from enum import Enum
		class Color(Enum):
		    RED = 1
		    GREEN = 2
		    BLUE = 3

Enum w/ data	enum Message {
	    Quit,
	    Move { x: i32, y: i32 },
	    Write(String),
	}
		# Use union types or classes
		from dataclasses import dataclass
		@dataclass
		class Move:
		    x: int
		    y: int

Generic	struct Container<T> {
	    value: T,
	}
		from typing import Generic, TypeVar
		T = TypeVar("T")
		class Container(Generic[T]):
		    value: T

Error Handling

Concept	Rust	Python

Result type	fn divide(a: i32, b: i32) -> Result<i32, String> {
	    if b == 0 {
	        Err("Division by zero".to_string())
	    } else {
	        Ok(a / b)
	    }
	}
		def divide(a, b):
		    if b == 0:
		        raise ValueError("Division by zero")
		    return a / b

Match Result	match result {
	    Ok(val) => println!("{}", val),
	    Err(e) => eprintln!("Error: {}", e),
	}
		try:
		    val = result()
		    print(val)
		except Exception as e:
		    print(f"Error: {e}")

Unwrap	let val = result.unwrap(); // panics if Err	val = result() # raises if error

Unwrap or	let val = result.unwrap_or(0);	try:
		    val = result()
		except:
		    val = 0

? operator	fn read_file() -> Result<String, io::Error> {
	    let content = fs::read_to_string("f.txt")?;
	    Ok(content)
	}
		def read_file():
		    # Exceptions propagate
		    with open("f.txt") as f:
		        return f.read()

Option	fn find(v: &[i32], target: i32) -> Option<usize> {
	    v.iter().position(|&x| x == target)
	}
		def find(v, target):
		    try:
		        return v.index(target)
		    except ValueError:
		        return None

Panic	panic!("Something went wrong!"); assert!(x > 0, "x must be positive");	raise RuntimeError("Something went wrong!") assert x > 0, "x must be positive"

Custom error	use std::error::Error;
	#[derive(Debug)]
	struct MyError;
	impl Error for MyError {}
		class MyError(Exception):
		    pass
		raise MyError("message")

File I/O

Operation	Rust	Python
Read file	use std::fs; let content = fs::read_to_string("file.txt")?;	with open("file.txt") as f: content = f.read()

Read lines	use std::io::{BufReader, BufRead};
	let file = File::open("f.txt")?;
	for line in BufReader::new(file).lines() {
	    let line = line?;
	}
		with open("file.txt") as f:
		    for line in f:
		        # line includes \n
		        line = line.strip()

Write file	use std::fs::File;
	use std::io::Write;
	let mut file = File::create("f.txt")?;
	file.write_all(b"text")?;
		with open("f.txt", "w") as f:
		    f.write("text")

Append	use std::fs::OpenOptions;
	let mut file = OpenOptions::new()
	    .append(true)
	    .open("f.txt")?;
		with open("f.txt", "a") as f:
		    f.write("text")

Check exists	use std::path::Path;
	Path::new("file.txt").exists()
		import os
		os.path.exists("file.txt")

Delete	std::fs::remove_file("file.txt")?;	import os
		os.remove("file.txt")

Read bytes	let bytes = fs::read("file.bin")?;	with open("file.bin", "rb") as f: bytes = f.read()

Common Patterns & Idioms

Pattern	Rust	Python

Iterators	vec.iter() // immutable
	vec.iter_mut() // mutable
	vec.into_iter() // consume
		iter(list)
		# Python iterators don't distinguish
		# mutability

Iterator chain	vec.iter()
	   .map(|x| x * 2)
	   .filter(|x| x > 10)
	   .collect()
		[x*2 for x in vec if x*2 > 10]
		# or
		list(map(..., filter(...)))

Range	0..10 // 0 to 9 0..=10 // 0 to 10	range(10) # 0 to 9 range(11) # 0 to 10

JSON	use serde_json;
	let v: Value = serde_json::from_str(data)?;
	let s = serde_json::to_string(&v)?;
		import json
		v = json.loads(data)
		s = json.dumps(v)

Sleep	use std::thread;
	use std::time::Duration;
	thread::sleep(Duration::from_secs(5));
		import time
		time.sleep(5)

Command line	use std::env;
	let args: Vec<String> = env::args().collect();
		import sys
		args = sys.argv

Modules	mod mymod;
	use mymod::function;
	pub fn public_fn() { }
		import mymod
		from mymod import function
		def public_fn():
		    pass

Tests	#[cfg(test)]
	mod tests {
	    #[test]
	    fn test_add() {
	        assert_eq!(2 + 2, 4);
	    }
	}
		import unittest
		class TestAdd(unittest.TestCase):
		    def test_add(self):
		        self.assertEqual(2+2, 4)

Docs	/// This function adds two numbers
	/// # Examples
	///
	```
	/// let result = add(2, 3);
	///
	```
	fn add(a: i32, b: i32) -> i32
		def add(a: int, b: int) -> int:
		    """Add two numbers.
		
		    Args:
		        a: First number
		        b: Second number
		    """

Key Philosophical Differences

Memory Safety: Rust guarantees memory safety at compile time through ownership system. Python uses garbage collection.

Performance: Rust compiles to native code (zero-cost abstractions). Python is interpreted and much slower.

Mutability: Rust variables are immutable by default (`let mut`). Python variables are always mutable.

Type System: Rust has strong static typing with inference. Python has dynamic typing with optional hints.

Error Handling: Rust uses `Result` and `Option` types (no exceptions). Python uses try/except.

Null Safety: Rust has no null - uses `Option`. Python has `None` (can cause errors).

Concurrency: Rust prevents data races at compile time. Python has GIL (limited parallelism).
Pattern Matching: Rust has exhaustive pattern matching. Python 3.10+ has limited match.
Macros: Rust has powerful compile-time macros. Python has runtime metaprogramming.
Zero-Cost: Rust abstractions have zero runtime cost. Python abstractions have overhead.