Purple Squirrel In a Nutshell

A Desktop Quick Reference

By Purple Squirrel Media

Preface

This book is designed to sit beside your keyboard. When you forget the exact syntax for a git rebase, need to look up an HTTP status code, or want to verify the correct way to handle ownership in Rust, this is the book you reach for.

Unlike tutorial-style books that walk you through concepts step by step, a Nutshell book assumes you already know what you want to do. You just need a quick reminder of exactly how to do it. Every page is optimized for fast lookup rather than linear reading.

The material is organized into three parts. Part I covers the tools you use every day: terminal commands, git workflows, package managers, and IDE shortcuts. Part II provides API reference material for REST conventions, GraphQL patterns, and authentication schemes. Part III offers language-specific cheatsheets for Python, TypeScript, Rust, and SQL.

Tables are used extensively throughout this book. This is intentional. When you need to quickly scan for a command or syntax pattern, a well-organized table beats prose every time.

Keep this book open in a split screen. Bookmark the sections you reference most. Annotate the margins with your own notes and shortcuts. A reference book earns its value through wear.

Part I. Developer Tools

The command line remains the most efficient interface for many development tasks. While graphical tools have their place, the terminal provides speed, scriptability, and precision that no GUI can match. This section covers the essential commands and workflows that professional developers use daily.

Chapter 1. Terminal and Shell

The shell is where developers spend much of their working lives. Whether you use bash, zsh, or fish, the core concepts remain the same. This chapter provides quick reference for the commands you'll use most frequently.

Navigation Commands

Moving through the filesystem efficiently is fundamental to productive shell work.

Command	Description
`pwd`	Print working directory
`ls -la`	List all files with details
`cd path`	Change directory
`cd -`	Return to previous directory

`cd ~`	Go to home directory

File Operations

These commands form the foundation of file manipulation from the command line.

Command	Description
`cp src dest`	Copy file
`cp -r src dest`	Copy directory recursively
`mv src dest`	Move or rename
`rm file`	Remove file
`rm -rf dir`	Remove directory recursively
`mkdir -p path`	Create nested directories
`touch file`	Create empty file or update timestamp

Text Processing

Unix philosophy centers on text as a universal interface. These commands let you inspect, search, and manipulate text streams.

Command	Description
`cat file`	Display file contents
`head -n 20 file`	Show first 20 lines
`tail -n 20 file`	Show last 20 lines
`tail -f file`	Follow file as it grows
`grep pattern file`	Search for pattern
`grep -r pattern dir`	Recursive search
`wc -l file`	Count lines

Pipes and Redirection

The power of Unix comes from composing simple tools into complex pipelines.

```
# Pipe output through multiple commands
cat file.txt | grep "error" | wc -l

# Redirect output to file
command > output.txt      # Overwrite
command >> output.txt     # Append
```

```
command 2>&1              # Redirect stderr to stdout
command &> output.txt     # Redirect both to file
```

Process Management

Understanding how to manage running processes is essential for development work.

Command	Description
`ps aux`	List all processes
`top` or `htop`	Interactive process viewer
`kill PID`	Terminate process gracefully
`kill -9 PID`	Force kill process
`command &`	Run in background
`jobs`	List background jobs
`fg %1`	Bring job to foreground

Chapter 2. Git Workflows

Git is the version control system that powers modern software development. This chapter covers everything from daily commands to advanced workflows.

Initial Configuration

Before using git, configure your identity and preferences.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
```

Daily Commands

These are the commands you'll use multiple times per day.

Command	Description
`git status`	Show working tree status
`git add .`	Stage all changes
`git add -p`	Interactive staging by hunk
`git commit -m "msg"`	Commit with message
`git push`	Push to remote
`git pull`	Fetch and merge

`git fetch`	Fetch without merge

Branching

Branches are lightweight in git. Use them liberally.

```
git branch                  # List branches
git branch feature          # Create branch
git checkout feature        # Switch to branch
git checkout -b feature     # Create and switch
git merge feature           # Merge into current
git branch -d feature       # Delete branch
```

Undoing Changes

Git provides multiple ways to undo work at different stages.

```
git checkout -- file        # Discard working changes
git reset HEAD file         # Unstage file
git reset --soft HEAD~1     # Undo commit, keep changes staged
git reset --hard HEAD~1     # Undo commit, discard changes
git revert <commit>         # Create inverse commit
```

Stashing

The stash lets you temporarily shelve changes without committing.

```
git stash                   # Stash current changes
git stash list              # List all stashes
git stash pop               # Apply and remove latest
git stash apply             # Apply but keep stash
git stash drop              # Remove without applying
```

Rebasing

Rebasing rewrites history for cleaner commit graphs.

```
git rebase main             # Rebase onto main
git rebase -i HEAD~3        # Interactive rebase last 3
git rebase --abort          # Cancel in-progress rebase
git rebase --continue       # Continue after resolving conflicts
```

Useful Aliases

Save keystrokes with git aliases.

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
```

```
git config --global alias.st status
git config --global alias.lg "log --oneline --graph --all"
```

Chapter 3. Package Managers

Every language ecosystem has its package manager. This chapter provides quick reference for the most common ones.

npm (Node.js)

Command	Description
`npm init -y`	Initialize new project
`npm install pkg`	Install dependency
`npm install -D pkg`	Install as dev dependency
`npm install -g pkg`	Install globally
`npm uninstall pkg`	Remove package
`npm update`	Update all packages
`npm run script`	Run package.json script
`npm list`	List installed packages
`npx command`	Run package without installing

pip (Python)

Command	Description
`pip install pkg`	Install package
`pip install -r requirements.txt`	Install from requirements
`pip uninstall pkg`	Remove package
`pip freeze > requirements.txt`	Export dependencies
`pip list`	List installed packages
`pip show pkg`	Show package info

Cargo (Rust)

Command	Description
`cargo new project`	Create new project
`cargo build`	Build project
`cargo build --release`	Build optimized

`cargo run`	Build and run
`cargo test`	Run tests
`cargo add pkg`	Add dependency
`cargo update`	Update dependencies

Homebrew (macOS)

Command	Description
`brew install pkg`	Install package
`brew uninstall pkg`	Remove package
`brew upgrade`	Upgrade all packages
`brew list`	List installed
`brew search term`	Search packages
`brew doctor`	Check system health
`brew cleanup`	Remove old versions

Chapter 4. IDE Shortcuts

Modern IDEs offer powerful features accessible through keyboard shortcuts. This chapter covers VS Code, the most widely used editor.

General Commands

Mac	Windows	Action
`Cmd+Shift+P`	`Ctrl+Shift+P`	Command Palette
`Cmd+P`	`Ctrl+P`	Quick Open file
`Cmd+,`	`Ctrl+,`	Settings
`Cmd+B`	`Ctrl+B`	Toggle sidebar
`Cmd+J`	`Ctrl+J`	Toggle terminal

Editing

Mac	Windows	Action
`Cmd+D`	`Ctrl+D`	Select next occurrence
`Cmd+Shift+L`	`Ctrl+Shift+L`	Select all occurrences
`Option+Up/Down`	`Alt+Up/Down`	Move line
`Shift+Option+Up/Down`	`Shift+Alt+Up/Down`	Copy line

`Cmd+/`	`Ctrl+/`	Toggle comment
`Cmd+Shift+K`	`Ctrl+Shift+K`	Delete line
`Cmd+Enter`	`Ctrl+Enter`	Insert line below

Navigation

Mac	Windows	Action
`Cmd+G`	`Ctrl+G`	Go to line
`F12`	`F12`	Go to definition
`Shift+F12`	`Shift+F12`	Find all references
`Cmd+Shift+O`	`Ctrl+Shift+O`	Go to symbol
`Cmd+\`	`Ctrl+\`	Split editor

Multi-cursor

Mac	Windows	Action
`Option+Click`	`Alt+Click`	Add cursor
`Cmd+Option+Up/Down`	`Ctrl+Alt+Up/Down`	Add cursor above/below
`Cmd+Shift+L`	`Ctrl+Shift+L`	Cursor at all occurrences

Part II. API Reference

Application programming interfaces define how software components communicate. Whether you're building APIs or consuming them, understanding the conventions and patterns is essential. This section covers REST, GraphQL, authentication, and rate limiting.

Chapter 5. REST Conventions

REST (Representational State Transfer) remains the dominant architectural style for web APIs. This chapter covers the conventions that make REST APIs predictable and easy to use.

HTTP Methods

Each HTTP method has specific semantics that clients and servers agree upon.

Method	Purpose	Idempotent	Has Body
`GET`	Read resource	Yes	No
`POST`	Create resource	No	Yes
`PUT`	Replace resource	Yes	Yes
`PATCH`	Partial update	Yes	Yes

`DELETE`	Remove resource	Yes	No

Status Codes

Status codes communicate the result of a request.

Code	Meaning
`200`	OK - Request succeeded
`201`	Created - Resource created
`204`	No Content - Success with empty body
`400`	Bad Request - Invalid input
`401`	Unauthorized - Authentication required
`403`	Forbidden - Insufficient permissions
`404`	Not Found - Resource doesn't exist
`409`	Conflict - State conflict
`422`	Unprocessable - Validation error
`429`	Too Many Requests - Rate limited
`500`	Server Error - Internal failure
`503`	Service Unavailable - Overloaded

URL Patterns

Consistent URL structures make APIs intuitive.

```
GET    /users              # List users
POST   /users              # Create user
GET    /users/:id          # Get user
PUT    /users/:id          # Replace user
PATCH  /users/:id          # Update user
DELETE /users/:id          # Delete user

GET    /users/:id/posts    # User's posts (nested resource)
```

Common Headers

```
Authorization: Bearer <token>
Content-Type: application/json
Accept: application/json
X-Request-ID: <uuid>
```

Chapter 6. GraphQL Patterns

GraphQL offers a query language for APIs that gives clients precisely the data they request.

Query Structure

```
query GetUser($id: ID!) {
  user(id: $id) {
    id
    name
    email
    posts(first: 10) {
      edges {
        node {
          id
          title
        }
      }
    }
  }
}
```

Mutations

```
mutation CreateUser($input: CreateUserInput!) {
  createUser(input: $input) {
    user {
      id
      name
    }
    errors {
      field
      message
    }
  }
}
```

Fragments

Fragments let you reuse field selections across queries.

```
fragment UserFields on User {
  id
  name
  email
  createdAt
}

query {
  me { ...UserFields }
```

```
  user(id: "123") { ...UserFields }
}
```

Variables

Variables are passed separately from the query.

```
{
  "id": "123",
  "input": {
    "name": "John",
    "email": "john@example.com"
  }
}
```

Chapter 7. Authentication

Authentication verifies identity. This chapter covers the most common authentication schemes.

JWT Structure

JSON Web Tokens consist of three base64-encoded parts: header, payload, and signature.

```
// Header
{ "alg": "HS256", "typ": "JWT" }

// Payload
{
  "sub": "user_123",
  "iat": 1609459200,
  "exp": 1609545600
}

// Usage
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

OAuth 2.0 Flow

```
1. Redirect user to provider
   → /authorize?client_id=X&redirect_uri=Y&scope=Z

2. User authorizes, provider redirects back with code
   ← /callback?code=ABC123

3. Exchange code for token
   POST /token { code, client_id, client_secret }
   → { access_token, refresh_token }

4. Use access token
   Authorization: Bearer <access_token>
```

API Keys

```
# Header (preferred)
X-API-Key: sk_live_abc123

# Query parameter (less secure, avoid if possible)
GET /api/data?api_key=sk_live_abc123
```

Chapter 8. Rate Limiting

Rate limiting protects APIs from abuse. Understanding the common patterns helps you build well-behaved clients.

Response Headers

```
X-RateLimit-Limit: 1000        # Maximum requests allowed
X-RateLimit-Remaining: 999     # Requests remaining
X-RateLimit-Reset: 1609459200  # Unix timestamp of reset
Retry-After: 60                # Seconds to wait (on 429)
```

Limiting Strategies

Strategy	Description
Fixed Window	N requests per time window
Sliding Window	Rolling window of time
Token Bucket	Tokens regenerate over time
Leaky Bucket	Constant output rate

Client-Side Handling

```python
import time

def api_call_with_retry(url, max_retries=3):
    for attempt in range(max_retries):
        response = requests.get(url)

        if response.status_code == 429:
            wait = int(response.headers.get('Retry-After', 60))
            time.sleep(wait)
            continue

        return response

    raise Exception("Rate limited after retries")
```

Part III. Language Cheatsheets

Different languages have different idioms and syntax. This section provides quick reference for the constructs you'll use most often in Python, TypeScript, Rust, and SQL.

Chapter 9. Python

Python emphasizes readability and expressiveness. This chapter covers the core syntax and patterns.

Data Types

```python
# Primitives
x = 42              # int
y = 3.14            # float
s = "hello"         # str
b = True            # bool
n = None            # NoneType

# Collections
lst = [1, 2, 3]              # list (mutable, ordered)
tpl = (1, 2, 3)              # tuple (immutable, ordered)
st = {1, 2, 3}               # set (mutable, unordered, unique)
dct = {"a": 1, "b": 2}       # dict (mutable, key-value)
```

Comprehensions

```python
# List comprehension
squares = [x**2 for x in range(10)]

# With condition
evens = [x for x in range(10) if x % 2 == 0]

# Dict comprehension
squared = {x: x**2 for x in range(5)}
```

Functions

```python
# Type hints
def greet(name: str) -> str:
    return f"Hello, {name}"

# Default arguments
def greet(name: str = "World") -> str:
    return f"Hello, {name}"

# Variable arguments
def func(*args, **kwargs):
    print(args)    # tuple of positional args
```

```
    print(kwargs)  # dict of keyword args

# Lambda
double = lambda x: x * 2
```

Classes

```
from dataclasses import dataclass

@dataclass
class User:
    name: str
    email: str
    active: bool = True

    def greet(self) -> str:
        return f"Hello, {self.name}"
```

Error Handling

```
try:
    result = risky_operation()
except ValueError as e:
    print(f"Value error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
else:
    print("Success!")
finally:
    cleanup()
```

Context Managers

```
# File handling
with open("file.txt", "r") as f:
    content = f.read()

# Custom context manager
from contextlib import contextmanager

@contextmanager
def timer():
    start = time.time()
    yield
    print(f"Elapsed: {time.time() - start:.2f}s")
```

Async/Await

```python
import asyncio

async def fetch_data(url: str) -> dict:
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.json()

async def main():
    results = await asyncio.gather(
        fetch_data("url1"),
        fetch_data("url2"),
    )
```

Chapter 10. TypeScript

TypeScript adds static typing to JavaScript. This chapter covers the type system and common patterns.

Type Annotations

```typescript
// Primitives
const num: number = 42;
const str: string = "hello";
const bool: boolean = true;
const arr: number[] = [1, 2, 3];

// Object types
interface User {
  id: string;
  name: string;
  email?: string;           // optional
  readonly createdAt: Date; // immutable
}

// Union types
type Status = "pending" | "active" | "closed";

// Generics
function first<T>(arr: T[]): T | undefined {
  return arr[0];
}
```

Type Guards

```typescript
function isString(value: unknown): value is string {
  return typeof value === "string";
}

if (isString(value)) {
```

```
  console.log(value.toUpperCase()); // type is string
}
```

Utility Types

```
Partial<T>     // All properties optional
Required<T>    // All properties required
Readonly<T>    // All properties readonly
Pick<T, K>     // Only specified keys
Omit<T, K>     // Exclude specified keys
Record<K, V>   // Object with key type K, value type V
```

Functions

```
// Typed function
function add(a: number, b: number): number {
  return a + b;
}

// Arrow function
const multiply = (a: number, b: number): number => a * b;

// Optional and default parameters
function greet(name: string, greeting?: string): string {
  return `${greeting ?? "Hello"}, ${name}`;
}
```

Classes

```
class User {
  constructor(
    public readonly id: string,
    public name: string,
    private email: string
  ) {}

  getEmail(): string {
    return this.email;
  }
}
```

Async/Await

```
async function fetchUser(id: string): Promise<User> {
  const response = await fetch(`/api/users/${id}`);
  if (!response.ok) {
    throw new Error("Failed to fetch");
```

```
  }
  return response.json();
}
```

Chapter 11. Rust

Rust guarantees memory safety without garbage collection. This chapter covers ownership, the type system, and common patterns.

Variables and Types

```
// Immutable by default
let x = 5;
let mut y = 10;  // mutable

// Type annotations
let num: i32 = 42;
let float: f64 = 3.14;
let s: String = String::from("hello");
let slice: &str = "hello";
```

Ownership

Rust's ownership system is unique and fundamental.

```
// Move semantics
let s1 = String::from("hello");
let s2 = s1;  // s1 is moved, no longer valid

// Clone for deep copy
let s1 = String::from("hello");
let s2 = s1.clone();  // both valid

// Borrowing
fn print_len(s: &String) {
    println!("{}", s.len());
}  // s is borrowed, not moved
```

Structs and Enums

```
struct User {
    name: String,
    age: u32,
}

impl User {
    fn new(name: String, age: u32) -> Self {
        Self { name, age }
    }
```

```
    fn greet(&self) -> String {
        format!("Hello, {}", self.name)
    }
}

enum Status {
    Active,
    Inactive,
    Pending { reason: String },
}
```

Pattern Matching

```
match value {
    0 => println!("zero"),
    1..=10 => println!("1-10"),
    _ => println!("other"),
}

// if let for single pattern
if let Some(x) = optional {
    println!("{}", x);
}
```

Error Handling

```
// Result type
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err("Division by zero".into())
    } else {
        Ok(a / b)
    }
}

// ? operator for propagation
fn calc() -> Result<f64, String> {
    let x = divide(10.0, 2.0)?;
    Ok(x * 2.0)
}
```

Iterators

```
let nums = vec![1, 2, 3, 4, 5];

let doubled: Vec<i32> = nums
    .iter()
```

```
    .map(|x| x * 2)
    .filter(|x| *x > 4)
    .collect();
```

Chapter 12. SQL

SQL is the universal language for relational databases. This chapter covers queries, joins, and schema management.

Basic Queries

```
-- Select
SELECT column1, column2 FROM table WHERE condition;
SELECT * FROM users WHERE active = true ORDER BY name;
SELECT DISTINCT category FROM products;

-- Insert
INSERT INTO users (name, email)
VALUES ('John', 'john@example.com');

-- Update
UPDATE users SET active = false
WHERE last_login < '2024-01-01';

-- Delete
DELETE FROM users WHERE id = 123;
```

Joins

```
-- Inner join (matching rows only)
SELECT u.name, o.total
FROM users u
INNER JOIN orders o ON u.id = o.user_id;

-- Left join (all from left table)
SELECT u.name, o.total
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;

-- Multiple joins
SELECT u.name, o.total, p.name
FROM users u
JOIN orders o ON u.id = o.user_id
JOIN products p ON o.product_id = p.id;
```

Aggregations

```sql
SELECT
    category,
    COUNT(*) as count,
    SUM(price) as total,
    AVG(price) as average,
    MAX(price) as max_price
FROM products
GROUP BY category
HAVING COUNT(*) > 5
ORDER BY total DESC;
```

Subqueries

```sql
-- In WHERE clause
SELECT * FROM users
WHERE id IN (SELECT user_id FROM orders WHERE total > 100);

-- Derived table
SELECT avg_order
FROM (
    SELECT user_id, AVG(total) as avg_order
    FROM orders GROUP BY user_id
) as user_averages
WHERE avg_order > 50;
```

Indexes

```sql
-- Create index
CREATE INDEX idx_users_email ON users(email);
CREATE UNIQUE INDEX idx_users_username ON users(username);

-- Composite index
CREATE INDEX idx_orders_user_date ON orders(user_id, created_at);

-- Drop index
DROP INDEX idx_users_email;
```

Transactions

```sql
BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;  -- or ROLLBACK to cancel
```

Purple Squirrel Media